Politechnika Warszawska

Warsaw University of Technology

https://repo.pw.edu.pl/

Tytuł / Title	Projekt i implementacja wybranych elementów silnika gry komputerowej / Design and implementation of selected elements of a computer game engine
Autor / Author	Babiński Michał
Promotor / Supervisor	Rokita Przemysław
Adres publikacji w Repozytorium URL / Publication address in Repository	https://repo.pw.edu.pl/info/bachelor /WUT1d939456e6a7473aa87568c952beb7aa/
Data opublikowania w Repozytorium / Deposited in Repository on	2023-11-14

Rok akademicki 2013/2014

Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

PRACA DYPLOMOWA INŻYNIERSKA

Michał Babiński

Projekt i implementacja wybranych elementów silnika gry komputerowej

Opiekun pracy:
prof. nzw. dr hab. inż. Przemysław Rokita

Ocena: .....

.....

Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Miejsce na zeskanowane lub wklejane zdjęcie

Kierunek: Informatyka

Specjalność: Inżynieria Systemów Informatycznych

Data urodzenia: 1989.05.10

Data rozpoczęcia studiów: 2010.02.22

Życiorys

Urodziłem się 10 maja 1989 r. w Otwocku, gdzie mieszkałem przez kolejne dwadzieścia trzy lata. Edukację rozpocząłem w Szkole Podstawowej nr 1 w Otwocku. Po ukończeniu gimnazjum nr 3, rozpocząłem edukację w Liceum Ogólnokształcącym im. Konstantego Ildefonsa Gałczyńskiego w klasie o profilu matematyczno-informatycznym. Zafascynowany tematyką programowania, w szczególności tworzenia gier komputerowych, jako ośrodek dalszego kształcenia wybrałem Wydział Elektroniki i Technik Informacyjnych Politechniki Warszawskiej.

.....
Podpis studenta

EGZAMIN DYPLOMOWY

Złożył egzamin dyplomowy w dniu ..... 2014 r.

z wynikiem .....

Ogólny wynik studiów: .....

Dodatkowe wnioski i uwagi Komisji: .....

.....

.....

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

STRESZCZENIE

Celem pracy było zaprojektowanie silnika gry trójwymiarowej, oraz implementacja w języku C++. Niniejsza praca poświęcona została omówieniu wymagań jakie stawiane są przed silnikiem gry komputerowej, oraz przegląd stosowanych rozwiązań. Silnik *Simple Project* powstał jako statyczna biblioteka języka C++, udostępniająca gotowe mechanizmy obiegu zdarzeń, komunikacji elementów gry, wykonywania kodu użytkownika, wyświetlania obrazu czy wielowątkowego ładowania zasobów jak tekstury czy modele trójwymiarowe. Ważnym elementem silnika stało się dążenie do optymalizacji algorytmów i funkcjonalności. Zastosowane rozwiązania jak drzewo ósemkowe, odrzucanie niewidocznych elementów czy zastosowanie mieszanych tablic wierzchołków spowodowało znaczący wzrost wydajności biblioteki, w stosunku do implementacji pozbawionej wyżej wymienionych funkcjonalności. Na potrzeby pracy, prócz silnika gry, powstał moduł generowania teksturowanego terenu z map wysokości. Jako element newralgiczny pod względem wydajności, został on w pełni zintegrowany z powyższymi funkcjonalnościami. Całość przedsięwzięcia została zweryfikowana poprzez stworzenie trzech gier komputerowych: *warcaby*, *wyścig*, *snajper*.

Słowa kluczowe: programowanie gier komputerowych, architektura silnika gry, grafika trójwymiarowa, OpenGL, C++, wzorce projektowe

---

Design and implementation of selected elements of a computer game engine

The aim of this work was to design a 3D game engine and implementation in C++. This work was devoted to the discussion of the requirements that are placed in front of the computer game engine, and an overview of solutions. Engine *Simple Project* was created as a static C++ library, that provides mechanisms are cycles of events, communication elements of the game, user code, display or multi-threaded resource loading textures and 3D models. An important part of the engine proved to be striving to optimize the algorithms and functionality. The solutions as octal tree, *View Frustum Culling* or the use of mixed vertex arrays resulted in a significant increase in performance compared libraries to implement devoid of the above mentioned functionality. For the purposes of work , in addition to the game engine , created a module to generate textured terrain height maps. As part of the sensitive in terms of performance , it has been fully integrated with the above functionality. The whole project has been verified through the creation of three video games: *checkers*, *race*, *sniper*.

Keywords: game development, game engine architekture, 3D graphics, OpenGL, C++, design patterns

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Spis treści

Cel pracy ... 6
Architektura silnika gry ... 7
Założenia techniczne ... 10
Założenia funkcjonalne ... 14
Moduły silnika ... 17
Rdzeń silnika ... 17
Zarządca obiektów ... 22
Zarządca zdarzeń ... 24
Moduł zasobów ... 27
Moduł renderowania ... 30
Moduł sceny ... 34
Moduł logowania błędów ... 36
Moduł profilowania kodu ... 39
Moduł funkcji i klas narzędziowych ... 42
Narzędzia dodatkowe ... 44
Mapy wysokości terenu ... 44
Teoria ... 44
Siatki trójkątów ... 46
Ukrywanie niewidocznych płaszczyzn ... 47
Teksturowanie ... 49
Implementacja i wyniki ... 49
Przykład gry ... 51
Gra warcaby ... 51
Tworzenie obiektów gry ... 52
Wbudowane prymitywy geometryczne ... 52
Obsługa zdarzeń ... 52
Animowanie ruchu metodą interpolacji liniowej ... 53
Skalowanie, przesuwanie i obracanie obiektów ... 53
Scena. Pobieranie obiektów sceny ... 54
Gra wyścig ... 55
Proxy tekstur i modeli 3D ... 56
Mapa wysokości ... 56
Sky Box (Sphere) ... 57
Pliki konfiguracyjne ... 57

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Rendering. Drzewo obiektów i Frustrum Culling 58
Gra snajper 59
Asynchroniczne wczytywanie zasobów 59
Rendering. OnGUI 60
Rendering. Przeźroczystość, maska 61
Kamera – ffp 62
Optymalizacje. Tablice wierzchołków 63
Zakończenie 64
Podsumowanie prac 64
Perspektywy rozwoju 65
Krytyczne spojrzenie 65
Bibliografia 67
Zawartość CD 68
Spis rysunków 68

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Cel pracy

Celem niniejszej pracy inżynierskiej było stworzenie działającego systemu informatycznego wspomagającego programistę w produkcji trójwymiarowych gier komputerowych. Silnik gry został zaimplementowany jako biblioteka języka C++ o nazwie *Simple Project*, i jak sugeruje jej nazwa, w zamyśle autora ma znaleźć zastosowania w małych niekomercyjnych projektach.

Główne założenia projektowe zostały podzielone na trzy grupy: łatwość użytkowania, uniwersalność, optymalne użycie sprzętu. Myśl o tych trzech elementach była obecna podczas całego procesu pracy, od projektowania do implementacji. Łatwość użytkowania silnika powinna implikować uzyskiwanie szybkich, ale i zadawalających rezultatów podczas tworzenia gier, uniwersalność polega na zaniechaniu wyboru konkretnego typu gry, który może być stworzony przy pomocy biblioteki. Wreszcie, to co odróżnia projekty pisane hobbistycznie od profesjonalnych - zadbanie o wydajność silnika. Mając na uwadze ograniczony czas oraz zespół projektowy złożony z samego autora, projekt nie ma w zamyśle pretendować do miana *profesjonalnego*, lecz jedynie podążać w tym kierunku. Duży wpływ na przebieg prac miała poznana w toku nauczania biblioteka QT oraz własne doświadczenia z komercyjnym silnikiem gry - Unity3D.

Jednym z pierwszych elementów projektowania był wybór elementów jakie powinien implementować silnik gry. Z pośród wielu funkcjonalności zostały wybrane niezbędne:

- Obsługa kontrolerów gry - mysz oraz klawiatura
- Wbudowane prymitywy geometryczne
- Wyświetlanie i teksturowanie obiektów
- Możliwość tworzenia, dodawania kodu wykowywanego przez obiekty gry
- Podstawowe operacje na obiektach: rotacje, skalowanie, przesuwanie we współrzędnych świata lub kamery
- Komunikacja pomiędzy obiektami sceny trójwymiarowej
- Wczytywanie modeli trójwymiarowych oraz tekstur
- Obsługa kamery
- *Skybox*

oraz dodatkowe:

- Animowanie rotacji, przesunięć i skalowania metodą interpolacji liniowej
- Dostęp do zasobów gry przez Pełnomocnika (ang. proxy)
- Wielowątkowe ładowanie zasobów
- Mapa wysokości (ang. height map)
- Drzewo ósemkowe oraz usuwanie niewidocznych obiektów (ang. viewing frustum culling)
- Przeźroczystości, maski
- Wolna kamera (ang. First-person view, Free Camera)
- Wbudowane mechanizmy do parsowania i odczytywania prostych tekstowych plików konfiguracyjnych

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Architektura silnika gry

Niniejszy rozdział zawiera wstępny zarys rozwiązań wykorzystanych w silniku powstałym na potrzeby pracy inżynierskiej. Początek rozdziału jest zwięzłym przeglądem możliwych do wykorzystania technologii, wzorców projektowych, oraz uzasadnieniem wyboru konkretnych rozwiązań. Następnie przedstawione zostały założenia techniczne, w których został opisany i uzasadniony wybór technologii, bibliotek oraz języka programowania. Podrozdziały zawierają informacje o wykorzystanym sprzęgu silnika z systemem operacyjnym oraz programami do tworzenia grafiki trójwymiarowej oraz opisują założenia funkcjonalne, jakie spełnić ma zaimplementowany silnik gry. W tym rozdziale opisana została mechanika wytwarzania gier komputerowych przy pomocy silnika. Rozdział wieńczy przegląd modułów oraz diagramy klas.

W dzisiejszych czasach silnik gry jest jednym z wielu elementów komercyjnych projektów. Do stworzenia gry komputerowej potrzeba szeregu narzędzi, jak programy graficzne do tworzenia modeli, tekstur, programy do tworzenia dźwięków, czy nawet aplikacje przy pomocy których buduje się rozbudowane listy dialogowe postaci, często o strukturach drzewiastych. Po części te narzędzia są dostarczane razem z rozwiązaniem: na przykład, komercyjny projekt Unity3D [1], dostarcza również program IDE *MonoDevelop*, oraz środowisko do budowy sceny 3D, które umożliwia wczytywanie modeli trójwymiarowych z zewnętrznych programów, łatwe rozmieszczanie elementów, kamer i świateł w widoku sceny. Jako, że w wymienionym silniku gry programiści implementują kod w językach C# lub JaveScript, środowisko można zintegrować z *Microsoft Visual Studio*. Z produktem Unity3D miałem przyjemność spotkać się w 2012 roku, pracując jako programista gier. *Unity Technologies*, jak można się domyślać, nie udostępnia kodu źródłowego produktu, jednak zaimplementowany na potrzeby tej pracy silnik, stara się wzorować na tym rozwiązaniu w kwestii mechaniki tworzenia gry od strony programistycznej. W Internecie dostępne są niezliczone przykłady i kursy dostarczane zarówno przez producenta jak i użytkowników. Na polskim rynku pojawił się również podręcznik przybliżający środowisko Unity3D [2].

Silnik *Simple Project* został zaimplementowany jako statyczna biblioteka języka C++. Podczas prac projektowych wykorzystałem doświadczenie jakie nabyłem pracując w środowisku *Unity3D*, oraz umiejętności zdobyte na studiach. Na kształt pracy znaczący wpływ miał mój kontakt z biblioteką QT którą poznałam na jednym z przedmiotów w toku nauczania. Podobnie jak w bibliotece wydanej przez *Qt Development Frameworks*, obiekt silnika gry jest *Singletonem* [3]. Jest to prosty sposób wykorzystania kompozycji, dzięki któremu w programie będzie obecny tylko jeden obiekt danej klasy. W przypadku zaimplementowanego silnika, takim *Singletonem* jest Rdzeń Silnika – klasa *SPApplication*. Kolejnym ciekawym udogodnieniem biblioteki QT jest drzewiasta struktura obiektów, zarówno okien jak i widgetów. Podobnie jak w językach Java czy C# twórcy QT zastosowali dziedziczenie wszystkich obiektów z jednej klasy, w tym przypadku *QObject* [4]. Specyfikuje ona między innymi wskaźnik na rodzica i listę dzieci. W przypadku zwalniania pamięci i niszczenia obiektu rodzica, pamięć zajmowana przez dzieci również zostanie zwolniona. Jest to użyteczne rozwiązanie pomagające uchronić się przed wyciekiem pamięci, które zaadaptowałem do powstałego silnika. Struktura drzewiasta okazała się użyteczna również w operacjach przesuwania, skalowania i obrotów obiektów. Elementy gry, będące dziećmi zostają poddane tożsamym operacjom co rodzice. Takie rozwiązanie lepiej odwzorowuje prawa rządzące światem materialnym.

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Ciekawym elementem silnika jest również mechanizm, ładowania zasobów *w locie*. Niektóre dzisiejsze gry, w szczególności z gałęzi sieciowych MMORPG [5], w których gracz przemierza olbrzymie obszary terenu, nie mogą pozwolić sobie na swobodne przechowywanie w pamięci operacyjnej wszystkich modeli, tekstur, poziomów, map wysokości. Przykładem takiej gry jest znana od lat gra *World Of Warcraft*, firmy *Blizzard Entertainment*. Obecnie zajmuje około 25GB przestrzeni dyskowej, i nawet przy dzisiejszym sprzęcie przechowywanie jednej czwartej danych w pamięci operacyjnej byłoby zadaniem karkołomnym. W przeciwieństwie do gier typu FPS [5] gdzie poziom podzielony jest na wiele etapów nie wymagających wczytywania i zwalniania kolejnych porcji danych w czasie rozgrywki, w grach MMORPG pominięcie mechanizmu wielowątkowego wczytywania zasobów, byłoby poważnym zaniechaniem gdyż, gracz zamiast beztrosko eksplorować świat, notorycznie spędzałby czas na oglądaniu paska postępu wczytywania mapy. Dodatkowo system wczytywania *w locie* stwarza kolejne możliwości. We wspomnianej już grze *Blizzard'a* gracz do gry potrzebuje jedynie zainstalowanej wersji *Core*, to znaczy pliku binarnego z kodem gry, gdyż mechanizm ładowania zasobów został powiększony o funkcjonalność pobierania odpowiednich plików od innych graczy w architekturze *P2P* [6]. Silnik WoW'a pobiera brakujące pliki od innych graczy jeśli nie znajdzie ich na komputerze klienta lub ich zawartość jest nieaktualna. W implementowanym silniku stworzyłem mechanizm pobierania zasobów *w locie*, jednak nie specyfikujący żadnych interfejsów do pobierania zasobów z zdalnych serwerów lub innych użytkowników silnika.

Gra komputerowa to przede wszystkim interaktywna zabawa. Interaktywność oparta na odbieraniu sygnałów od kontrolerów gry oraz dynamiczna reakcja na zdarzenia jest wpisana w każdą dzisiejszą grę komputerową. Istnieje kilka pomysłów na obsługę zdarzeń w silniku. Wspomniana biblioteka QT realizuje mechanizm tzw. *sygnałów i slotów* [7]. Jest on rozwinięciem często stosowanych *wywołań zwrotnych* [8] (ang. callback). Sygnały i sloty zapewniają bezpieczeństwo typów przekazywanych argumentów, jednak są wolniejsze. Innym sposobem obsłużenia zdarzeń jest *Nasłuchiwacz* [9] (ang. Action Listener). Obiekt nasłuchuje zdarzeń i w wyniku zmiany stanu kontrolerów wykonuje odpowiednie funkcje zarejestrowane przez użytkowników. Poniższy przykład ilustruje użycie *Nasłuchiwacza* w języku *ActionScript 3.0*:

```
function wyswietlWiadomosc(e:Event):void
{
	trace("Właśnie klikasz myszką");
}
Moj_przycisk.addEventListener(MouseEvent.CLICK, wyswietlWiadomosc);
```

W implementacji silnika wybrałem właśnie ten sposób obsługi zdarzeń. Architektura tego rozwiązania opiera się na wzorcu *Obserwator* [3]. W przygotowanej implementacji zapewniłem bezpieczeństwo typów, poprzez odpowiednią kompozycję klas dziedziczących z klasy *SPEvent* oraz poszczególnych Obserwatorów.

Tworzenie kodu użytkownika jest kolejnym zagadnieniem godnym uwagi. W większości projektów, również w wymienionej bibliotece QT oraz Unity3D funkcjonalności standartowych elementów są rozszerzane przez dziedziczenie. Przykładowy obiekt, aby mógł zaistnieć w scenie a biblioteka mogła kierować do niego zdarzenia musi dziedziczyć po klasie QObject w przypadku QT oraz po *MonoBehaviour* w przypadku Unity3D. Jednakże w językach obiektowych rozszerzanie nie musi odbywać się przez dziedziczenie. Czasami wręcz nie chcemy aby obiekt dziedziczył niektórych pól i zachowań. Rozszerzanie funkcjonalności

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

możemy osiągnąć przez kompozycję, na przykład jeśli zdecydujemy się na wykorzystanie wzorca *Dekorator* [3]. Pozwala on na dowolne rozszerzanie funkcjonalności klas, i jest ciekawym rozwiązaniem. Poniżej przedstawiono przykład użycia wzorca *Dekorator*:

```
GameObject pistolet = new PoruszanieDekorator (
                new WyswietlanieDekorator(new GameObject()));
pistoler.WykonajAkcje();
```

Powyższy kod udekoruje *pistolet* w możliwość przesuwania na przykład po wciśnięciu klawisza klawiatury oraz wyświetlenie modelu obiektu na ekranie. Wzorzec dekorator stwarza jednak problem, związany z kolejnością wykonywania funkcjonalności w jakie udekorowaliśmy obiekty. Jeśli Dekorator *Poruszanie* powoduje odczytanie stanu klawiszy i przesunięcie obiektu, natomiast Dekorator *Wyświetlanie* wyrenderowanie obiektu to łatwo można wpaść w pułapkę. Poniższy kod pokazuję przypadek w którym występuje rozsynchronizowania sceny:

```
GameObject zajac = new WyswietlanieDekorator (
              new PoruszanieDekorator (new GameObject()));

GameObject zolw = new PoruszanieDekorator (
              new WyswietlanieDekorator (new GameObject()));
```

W obu przypadkach wciśnięcie strzałki spowoduje przesunięcie zająca oraz żółwia w prawą stronę, poprzez dodanie odpowiedniej wartości do współrzędnej x każdego z obiektów. Funkcja renderująca obiera pozycje x i wysyła listę wierzchołków do karty graficznej w celu wyrenderowania sceny. Ze względu na inną kolejność dekorowania żółwia i zająca, żółw prześcignie zająca w pierwszej klatce ponieważ został przesunięty, a następnie wyświetlony na ekranie. Zając odwrotnie, została wyświetlona jego poprzednia pozycja a następnie odświeżona wartość x spowodowana wciśniętym przyciskiem. Te niejednoznaczności spowodowały rezygnację z tego skąd inąd bardzo ciekawego rozwiązania.

Nieczęstym, ale świadczącym o profesjonalizmie, elementem systemu informatycznego jest dynamiczna analiza kodu, znana jako *Profiler* [10]. Częstym problemem w programowaniu są obszary które odznaczają się długością wykonania na tle reszty kodu. Zadaniem *Profilera* jest pomiar długości wykonania funkcji w celu namierzenia potencjalnych obszarów optymalizacji. Środowisko Unity3D udostępnia możliwość analizy kodu programem profilującym [11], podobnie jak większość programów IDE. Profiler mógłby być kolejnym przykładem klasy typu *Singleton*, dostępnej z każdego miejsca kodu, oraz dodatkowo bezpieczną w kontekście wywołań w aplikacjach wielowątkowych. *Profiler* zazwyczaj posiada strukturę drzewiastą, a przykład takiej implementacji został opisany w *Perełkach Programowania* [10].

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Założenia techniczne

Większość obecnych silników gier, jak Source, Unreal Engine czy Unity3D, została zaimplementowana w języku C++. Fakt ten nie determinuje jednak języka w którym będą programowane gry komputerowe. Przykładem takiego odstępstwa jest Unity 3D. Choć silnik został napisany w języku C++ to językiem programowania skryptów które wykonują poszczególne obiekty w grze jest Boo, JavaScript lub C# [14]. Trudno jednak obrać takie rozwiązanie w obrębie pracy inżynierskiej, ponieważ wymaga ono implementacji odpowiedniego interfejsu silnika gry dla innych języków programowania. Warto jednak nadmienić, iż w silniku Unity3D, wybór języka jest dowolny – istnieje możliwość pisania skryptów w różnych językach w obrębie tego samego projektu, oraz wywoływanie funkcji czy dostęp do zmiennych języka C# z JavaScript i odwrotnie. Pisanie projektu w dwóch językach naraz powoduje jednak niedogodności związane z kolejnością kompilowania kodu.

Kolejnym ważnym założeniem jaki postawiłem przed implementacją było stosowanie wzorców projektowych oraz budowa przemyślanych połączeń pomiędzy obiektami. W implementacji wykorzystane zostały wszystkie dobrodziejstwa paradygmatu obiektowego jak hermetyzacja kodu, polimorfizm czy dziedziczenie. W ograniczonym stopniu zastosowany został globalny dostęp do obiektów przez funkcje statyczne. W pracy zostały użyte znane i dość popularne wzorce projektowe. Obsługa zdarzeń krążących w bibliotece silnika została zaimplementowana jako Obserwator [3], moduł dynamicznego ładowania i zwalniania zasobów korzysta z wzorca Pełnomocnika (ang. proxy) [3], dodawanie obiektów do sceny zostało zaimplementowane jako Fabryka [3]. W projekcie istnieje dość nietypowe zastosowanie wzorca Fasada [3] do ograniczania widoczności pól klasy w niektórych przypadkach. Użycie powyższych wzorców projektowych nigdy nie było celem samym w sobie. Problemy jakie napotkałem w implementacji często zachęcały do skorzystania z przemyślanych i gotowych rozwiązań, zwiększających szanse powodzenia projektu. Projektowanie aplikacji było ważnym oraz długotrwałym procesem, mającym wpływ na wyniki implementacji. W dalszej części pracy umieściłem diagramy klas w języku UML, które powstawały wraz z procesem implementacji, oraz przykłady kodu obrazujące wykorzystanie wzorców w praktyce.

Wybór biblioteki graficznej został dokonany jeszcze przed wyborem języka programowania. Obecnie biblioteki DierctX oraz OpenGL mają zbliżone funkcjonalności oraz API w językach C, C++ C# i innych. Wybór został dokonany przez wzgląd na łatwość użycia i doświadczenie autora. Pod tym kryterium wybrałem bibliotekę OpenGL z którą zapoznałem się w toku studiów oraz własnych projektów. Warto jednak zauważyć, iż w odróżnieniu od DirectX, w OpenGL nie jest wymagany najnowszy system Windows do działania wszystkich funkcjonalności biblioteki. Jako, że w perspektywie dalszych prac planowany jest rozwój na platformę LINUX nie mniej ważnym autem OpenGL jest możliwość przenoszenia kodu i kompilacji na innych platformach. W komercyjnych projektach jak Unity3D, czy silnik Source, DirectX używany jest jako API graficzne na platformach Microsoft oraz Xbox, a w przypadku reszty platform używany jest właśnie OpenGL. Twórcy sprzętu, w swoich produktach, w pełni implementują możliwości zarówno OpenGL jak i DirectX. Jako ciekawostkę podać można, iż swój udział przy tworzeniu standardu OpenGL miał również jeden z absolwentów Wydziału Elektroniki i Technik Informacyjnych Politechniki Warszawskiej - Marek Hołyński [31].

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Założeniem dotyczącym sprzęgu z warstwą systemu operacyjnego był wybór natywnych interfejsów (API) poszczególnych systemów oraz rezygnacja z bibliotek graficznych jak *Allegro* [15] czy *SDL* [16]. W ramach tej pracy operacje tworzenia okna, pobierania komunikatów od użytkowników, tworzenie wątków zostały zaimplementowane w Windows API.
W dalszych pracach nad projektem rozwiązanie dla systemów Linuksowych zostanie zaimplementowane w *X-Window* oraz standardzie *POSIX*. Takie rozwiązanie często spotykane jest w świecie gier, jest lepiej dopasowane i optymalne pod wybraną platformę, powoduje niestety konieczność napisania własnego interfejsu warstwy abstrakcji systemu operacyjnego, oraz implementacji interfejsu pod poszczególne systemy operacyjne. W stworzonym na potrzeby pracy inżynierskiej silniku, sprzęg z systemem operacyjnym jest realizowany przez moduł *OS_Layer*, dostarczający między innymi klasy *SPMutex, SPWindow* czy *SPThread*.

Jednym z wymagań dotyczących obiektów sceny jest przechowywanie ich wskaźników w strukturze drzewiastej. Korzeniem drzewa obiektów jest scena, która tym samym dziedziczy tę funkcjonalność z obiektu *SPSceneElement*. Operacje przesuwania, skalowania i obrotów są propagowane rekurencyjnie na dzieci danego obiektu. Takie rozwiązanie jest standardem projektów informatycznych jak np. *Adobe Flash*, wspominana wielokrotnie biblioteka QT oraz wielu innych. Relacje rodzic-dziecko zostały również wykorzystane do zwalniania pamięci podobnie jak ma to miejsce w bibliotece QT [17]. W poniższym przykładzie, nadawanie hierarchii obiektów odbywa się poprzez funkcję *AddChild*, powoduje ona dodanie elementu do sceny, ustawienie wskaźnika rodzica w nowo utworzonym Kole oraz dodanie koła do listy dzieci obiektu Samochód. W przypadku niepowodzenia operacji, na przykład w wyniku podania istniejącego już w aplikacji identyfikatora elementu funkcja zwraca zero, a obiekt jest zwalniany:

```
Kolo *lewe_tylne = static_cast<Kolo*>( Samochod->AddChild("k_l_t", new Kolo()));
if(lewe_tylne == NULL)
{
	// obsługa
}
```

Obiektem wszelkich starań optymalizacyjnych jest wyświetlanie grafiki. Procesor w dzisiejszych grach nie jest tak znacznie obciążony. W zaimplementowanym silniku jego rola ogranicza się do wykonania pętli gry, która składa się z odświeżania licznika czasu, wykonania kodu obiektów (który powinien być zwięzły), ładowania zasobów z dysku – który jest urządzeniem rzędy wielkości wolniejszym od procesora. Taki stan rzeczy nie odnosi się jednak do karty graficznej. Należy mieć na uwadze fakt ogromnej liczby operacji wykonywanych przez kartę graficzną, szczególnie mnożenia macierzy, określania głębokości pikseli, teksturowania, rozwiązywania cieniowania dla płaszczyzn których liczba osiąga nawet setki tysięcy w obrębie sceny. W silnikach gier ogranicza się ilość operacji karty graficznej przez przechowywanie obiektów w drzewach ósemkowych [18]. Scena dzielona jest rekurencyjnie na osiem sześcianów do ustalonej głębokości drzewa. Podczas renderowania rozwiązuje się test widoczności bryły, i w zależności od wyniku testu, odrzuca się przeglądanie całej gałęzi lub algorytm zagłębia się dalej renderując kolejne elementy [19].

Kolejną optymalizacją jest odrzucanie płaszczyzn odwróconych tyłem polegająca na badaniu wektorów normalnych płaszczyzn. Biblioteka OpenGL może wykonywać taki test jeśli zostanie uruchomiony:

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

```
glEnable ( GL_CULL_FACE );
glCullFace ( GL_BACK );
```

Odrzucanie ścian odwróconych tyłem do obserwatora zmniejsza ilość obliczeń związanych z mnożeniem macierzy wierzchołków ale przede wszystkim wyznaczania cieni.

W celu zwiększenia wydajności silnik wykorzystuje przeplatane tablice wierzchołków. Są one rzędy wielkości wydajniejsze niż wykorzystywanie funkcji *glVertex3f*. Poniższe przykłady prezentują sposoby renderowania. Przykład drugi pozwalał na osiąganie 30 klatek na sekundę przy wyświetlanej ilości wierzchołków nie przekraczającej 10 000, natomiast zastosowanie tablic przeplatanych podniosło ten próg do 160 000 wierzchołków. W projekcie zastosowano przeplatane tablicę typu *GL_T2F_N3F_V3F*, gdzie element tablicy składa się z dwóch liczb zmiennoprzecinkowych pojedynczej precyzji opisujących współrzędne teksturowania, trzech opisujących wektor normalny oraz trzech współrzędnych.

```
// (1) niewydajne renderowanie - rozwiązanie najgorsze, dużo wywołań glBegin
	for( ... )
	{
		glBegin();
		renderujtrojkat();
		glEnd();
	}
// (2) ograniczenie wywołań glBegin
	glBegin();
	for( ... )
	{
		renderujtrojkat();
	}
	glEnd();
// (3) wykorzystanie tablic wierzchołków

	glInterleavedArrays(GL_T2F_N3F_V3F,	// typ elementu
		0,				// offset
		(*obj).table			// wskaźnik na tablicę
	);
	glDrawArrays(GL_TRIANGLES, 0, (*obj).size * 3);
```

Kolejną optymalizacją było stworzenie dwóch kolejek renderowania. Jedna jest używana do tworzenia obrazu na ekranie, druga używana do rozstrzygania zdarzenia kliknięć na obiekty. W momencie wciśnięcia lub zwolnienia przycisku myszy przez gracza, renderowane są jedynie obiekty które zostały zarejestrowane jako nasłuchiwacze zdarzeń *OnPress*, *OnRelease*. Dodatkowo renderowanie w trybie *GL_SELECT* odbywa się w zubożałej formie, bez świateł, cieni oraz teksturowania.

Istnieją jeszcze inne możliwości zwiększania wydajności renderowania, jak siatki trójkątów, czy metoda poziomu szczegółowości (ang. level of detail) [20]. Wiele zostało opisanych i umieszczonych w Internecie wraz z przykładami [32].

Ostatnim, lecz nie mniej ważnym niż poprzednie, założeniem było wyspecyfikowanie interfejsów umożliwiających wczytywanie modeli trójwymiarowych oraz tekstur do silnika gry. Na potrzeby pracy zastała zaimplementowana funkcjonalność odczytu plików w prostym standardzie Wavefront [33] obsługiwanym przez większość programów graficznych.

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Standard składa się z dwóch plików – opisującego geometrie oraz materiały. Poszczególne wierzchołki modelu są zapisywane w postaci kolejnych wierszy z indeksem *v* natomiast płaszczyzny opisane są indeksem *f*. Poniżej zamieściłem fragment zawartości pliku opisującego sześcian:

```
mtllib szescian.mtl
o Cube
v 1.0 -1.0 -1.0
...

vt 1.0 0.0
...

usemtl Material
f 2/1 1/2 3/3 4/4
...
```

Na początku pliku definiowana jest nazwa pliku materiałów, nazwa obiektu oraz listy wierzchołków i współrzędnych teksturowania *vt*. Każda płaszczyzna może być opisana trójkątem lub kwadratem, w zależności ile elementów będzie zawierał wiersz *f*.
W powyższym przykładzie płaszczyzna jest kwadratem o odpowiednim indeksie wierzchołka i indeksie współrzędnej teksturowania. Standard przewiduje również zapis

```
f 2/1/3 1/2/2 3/3/3 4/4/4
```

gdzie liczba po drugim znaku „/” jest indeksem wektora normalnego (*v/vt/vp*). Indeksacja jak widać na przykładzie nie zaczyna się od cyfry 0 ale 1.

Dodatkowo na potrzeby pracy zaimplementowano wczytywanie 24-bitowych bitmap. Implementacja umożliwia odczyt bitmapy posiadającej standardowy 54 bajtowy nagłówek. Ciekawostką implementacyjną jest fakt, iż w standardzie bitmapy trzy kolejne bajty odpowiadają kolorom: niebieski, zielony, czerwony – BGR, odwrotnie niż w większości formatów graficznych. Dodatkowo bitmapy o wierszach nie będących wielokrotnością liczby cztery, są dopełniane zerami [34].

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Założenia funkcjonalne

Główną funkcjonalnością silnika gry, jest możliwość stworzenia gry komputerowej. Jednak sposób w jaki tworzy się gry nie jest już taki oczywisty. Końcowy produkt jest owocem pracy przedstawicieli z różnych dziedzin. Gra to programowanie, grafika, muzyka, fabuła, dialogi, kompozycja poziomów. Trudno wyobrazić sobie program który będzie udostępniał możliwości rozwoju gry w wszystkich wymienionych kierunkach jednocześnie. Nasuwa się pytanie, czy wygodnie byłoby programiście pracować w środowisku które umożliwia wszystkie te opcje. Jak wspomniałem Unity3D specyfikuje programy odpowiednie dla poszczególnych twórców. Programista chętnie wybierze IDE jako środowisko pracy, architekt sceny raczej niechętnie ustawi obiekty sceny korzystając z plików konfiguracyjnych, czy języka programowania lecz wybierze program umożliwiający podgląd sceny trójwymiarowej i wprowadzanie zmian na bieżąco. Sam program umożliwiający edycję sceny nie posiada tak rozbudowanych opcji jak programy graficzne PhotoShop, 3dsMax które zostaną wybrane przez grafików.

Z racji wyboru studiów informatycznych obiektem moich zainteresowań jest przede wszystkim wyspecyfikowanie sposobu tworzenia gry przez programistów. Sprzęg zaimplementowanego silnika z programami graficznymi następuje poprzez wymienione już pliki standardu Wavefront i bitmapy. Takie rozwiązanie nie jest jednak idealne, obiekty wczytywane, są okrojone do możliwości wybranego pliku, rozszerzenia. Braki danego standardu muszą zostać uzupełniane już po stronie silnika, przykładem może być konfiguracja mgły, kamer i świateł ustawiana przez programistę lub w odpowiednim programie projektowym. W pełni użyteczny program służący do dekoracji sceny powinien umożliwiać interfejs do silnika gry. Za taki interfejs można uznać pliki konfiguracyjne, XML czy skrypty Lua [21]. Architekt sceny swobodnie dekoruje scenę, a środowisko projektowe tworzy plik XML opisujący scenę. Jednak możliwość wczytywania definicji sceny podczas działania skompilowanego kodu nie spełnia wszystkich wymagań. Problemem tego mechanizmu jest niemożność dodania do sceny obiektów nie istniejących w bibliotece silnika, w szczególności klas napisanych przez użytkowników. Ten problem może zostać rozwiązany podobnie jak w programach IDE obsługujących tak zwanego *Designera*. Przykładem takiego rozwiązania jest sposób tworzenia aplikacji webowych lub okienkowych w *Visual Studio*. Użytkownik ma możliwość dodawania kolejnych obiektów okna poprzez przenoszenie i upuszczanie kontrolek w wybrane miejsce. Środowisko *Visual Studio* jednocześnie dodaje definicje obiektów w pliku o rozszerzeniu *.Designer* bez wiedzy użytkownika. Do takiego rozwiązania w przypadku C# stosowany jest mechanizm *partial* umożliwiający częściową definicję klasy w różnych plikach. Poniższy przykład przedstawia efekt umieszczenia przycisku w trybie Designera:

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

```
partial class Form1
{
	#region Windows Form Designer generated code
	private void InitializeComponent()
	{
		this.button1 = new System.Windows.Forms.Button();
		this.SuspendLayout();
		//
		// button1
		//
		this.button1.Location = new System.Drawing.Point(43, 66);
		this.button1.Name = "button1";
		this.button1.Size = new System.Drawing.Size(75, 23);
		this.button1.TabIndex = 0;
		this.button1.Text = "button1";
		this.button1.UseVisualStyleBackColor = true;
	}
	#endregion
}
```

Zmiana tekstu wyświetlanego na przycisku spowoduje zmianę w ciele funkcji inicjalizującej którą wygenerowało środowisko, bez udziału użytkownika. Tak powstała klasa składająca się z kodu stworzonego przez użytkownika, oraz kodu wygenerowanego automatycznie przez *Designer*, zostanie kompilowana. Z takiego rozwiązania wynika cenna właściwość również dla silnika gry: użytkownik umieszczający przycisk w oknie, nadający mu wygląd i wymiary, nie musi posiadać wiedzy programisty. W kontekście silnika gry, automatyczny generator kodu powinien stworzyć funkcję inicjalizującą każdą scenę, tworząc obiekty napisane przez programistów, oraz odpowiednie połączenia hierarchii rodzic-dziecko. Gotowy kod zostaje skompilowany, a gra uruchomiona. Praca programisty ogranicza się więc do dwóch zadań:

- Tworzenie definicji klas oraz ich funkcji
- Definiowanie obsługi zdarzeń kierowanych do danego obiektu klasy

Wykonywanie kodu obiektów napisanych przez użytkowników odbywa się poprzez implementacje wirtualnych funkcji obsługi zdarzeń obiektu *SPSceneElement*. Użytkownik ma możliwość obsługi zdarzeń takich jak wciśnięcie klawisza, ale też utworzenie obiektu czy wygenerowanie nowej klatki animacji. Kod wykonywany w obsłudze tych zdarzeń może dodawać, usuwać, modyfikować inne obiekty sceny oraz sterować modułami renderowania czy zarządzania zasobami. Ze względu na konieczność synchronicznego wykonania kodu z renderowaniem sceny, kod obsługi zdarzeń i wyświetlania wykonywany jest w jednym wątku. Takie rozwiązanie jest mniej problematyczne w implementacji ale obarczone niedogodnościami. Zapętlenie w funkcji napisanej przez programistę gry, spowoduje zatrzymanie całej aplikacji. Wykonanie długiej pętli w obsłudze zdarzenia będzie zmniejszać ilość klatek wyświetlanych w jednostce czasu.
Umieszczenie obsługi zdarzeń i renderowania sceny w jednym wątku prowadzi również do problemów długich operacji związanych z dostępem do pamięci dyskowej lub zasobów sieciowych. W sytuacji gdy projektant po naciśnięciu klawisza chce aby do sceny został dodany obiekt posiadający model trójwymiarowy z nałożonymi teksturami, z dużą pewnością obniży częstotliwość wyświetlania obrazu, przez zatrzymanie całej pętli na czas odczytów z dysku. W odpowiedzi na problemy długich operacji został zaimplementowany mechanizm asynchronicznego ładowania zasobów w odrębnym wątku.

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Ostatnim ale równie ważnym założeniem implementacyjnym było zaprojektowanie prostego interfejsu który nie odstraszy użytkowników pragnących wykorzystać silnik do stworzenia wymarzonych gier. Umieszczony poniżej kod przedstawia z jaką łatwością można stawiać pierwsze kroki w pisaniu gry z użyciem stworzonego przeze mnie silnika:

```
int main()
{
	SPApplication *app = SPApplication::GetInstance();
	SPModelProxy * modelProxy = SPModelProxy::GetInstance();

	Bron *p1 = new Bron();

	p1->DisplayModel =
		modelProxy->LoadResource("mp-5.obj", 0,  false).Resource;
	p1->MoveElement(Point3D(2.0f,0.0f,2.0f));

	Camera* c1 = s1->AddCamera("kamera1", new Kamera());
	app->Events.EnterFrame.AddHandler(&SPSceneElement::OnEnterFrame,c1);

	return App->Run();
}
```

Powyższy kod pobiera instancje aplikacji, oraz obiektu zarządzającego modelami trójwymiarowymi. Broni przypisany jest model wywołaniem blokującej metody z parametrem *false*, alternatywnie do funkcji można przekazać wskaźnik na obiekt, a zasób będzie dostarczony w parametrze zdarzenia wczytania zasobu. Wywołanie funkcji *Run* rozpoczyna wykonywanie pętli głównej gry. W powyższym przykładzie obiekty klasy Broń, Kamera są przykładami obiektów klas stworzonych przez programistę i nie są częścią biblioteki silnika. Do obiektów aktywnej sceny kierowane są wszystkie zdarzenia i wykowany jest kod ich obsługi. Do programisty należy stworzenie odpowiednich skryptów opisujących mechanikę rozgrywki.

Zaimplementowany na potrzeby tej pracy silnik z perspektywy programisty jest biblioteką języka C++, bazująca na wzorcu *Singleton* podobnie jak biblioteka QT [17]. W dalszej części rozdziału zostaną przedstawione szczegóły implementacyjne poszczególnych modułów, oraz przykłady użycia biblioteki.

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Moduły silnika

Silnik gry został podzielony na dziewięć modułów z zachowaniem zasady enkapsulacji kodu. Interfejsy jakie udostępniają poszczególne klasy są proste i ograniczają się do kilku metod. Celem projektowanego silnika była łatwość użycia dostępnych klas w celu zachęcenia programistów do korzystania z silnika. We wszystkich przypadkach klasy powiązane są ze sobą w przemyślany sposób, likwidując do niezbędnego minimum globalny dostęp do poszczególnych modułów. Prócz wymienionego wczesniej obiektu aplikacji opisanego klasą *SPApplication*, w kilku przypadkach zastosowany został statyczny dostęp do obiektów wykorzystujący wzorzec *Singleton*.

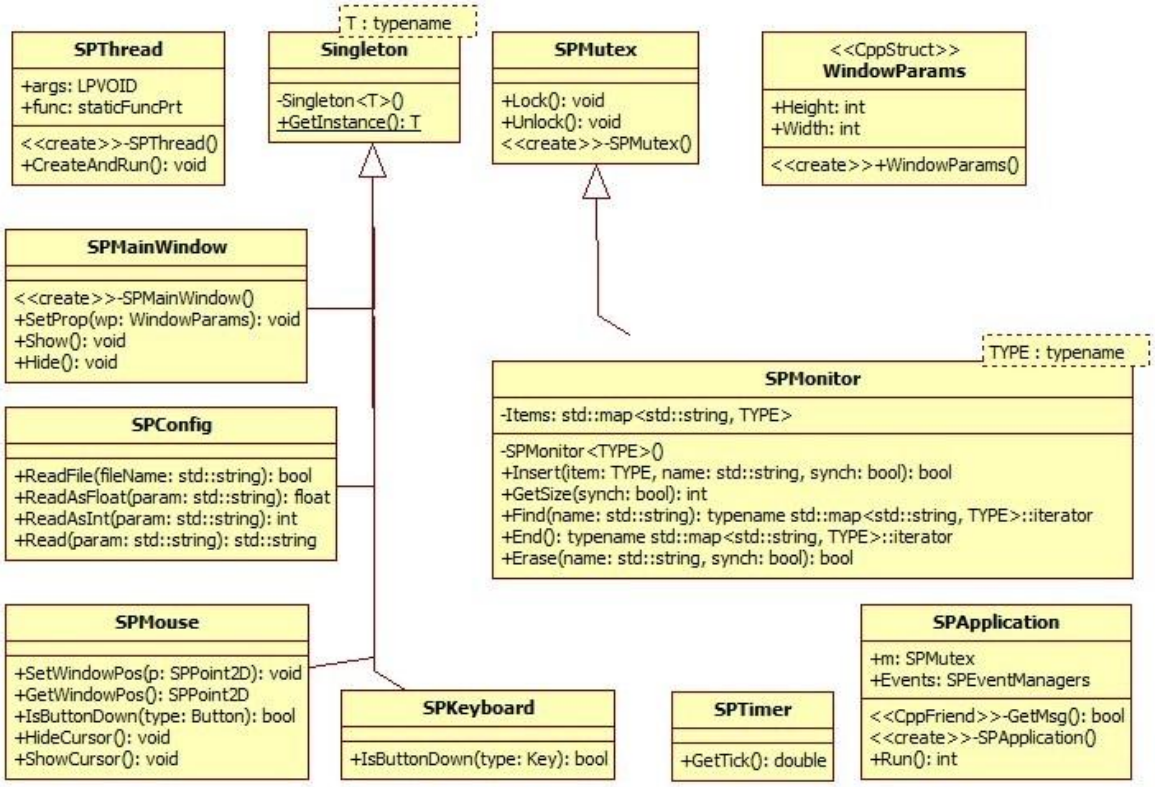
Poniższy wykaz przedstawia elementy silnika:

- Rdzeń silnika i moduł warstwy systemu operacyjnego
- Zarządca obiektów
- Zarządca zdarzeń
- Moduł zasobów
- Moduł renderowania
- Moduł sceny i obiektów
- Moduł logowania błędów
- Moduł profilowania kodu
- Moduł funkcji i klas narzędziowych

Rdzeń silnika

Implementacja rdzenia silnika została umieszczona w plikach *SPApp.h, .hpp i .cpp,* oraz *OS_Layer.h* i *.cpp*. Poniższy diagram przedstawia klasy modułu:

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29



Rys. 1 Diagram klas modułu Rdzeń Silnika

Założeniem projektowym budowy rdzenia była minimalizacja kodu, ograniczenie ilości klas składających się na moduł oraz udostępnienie interfejsów do innych modułów poprzez obiekt rdzenia. Programiście został udostępniony globalny dostęp do obiektu aplikacji, jednakże środowisko dba aby obiekt rdzenia nie mógł zostać skopiowany lub aby nie istniały w systemie dwa obiekty rdzenia. Dodatkowo nie może wystąpić sytuacja w której programista może podmienić obiekt rdzenia inną jego instancją. Z pomocą w rozwiązaniu powyższych problemów przychodzi wzorzec projektowy Singleton [23]:

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

```
template<typename T> class Singleton
{
private:
	static T* Instance;
	static SPMutex m;
	~Singleton<T>() {};
	Singleton( const Singleton & s );
protected:
	Singleton<T>() {};
public:
	inline static T* GetInstance()
	{
		Singleton<T>::m.Lock();
		if(Singleton<T>::Instance == 0)
		{
			Singleton<T>::Instance = new T();
		}
		Singleton<T>::m.Unlock();
		return Singleton<T>::Instance;
	};
};

// main.cpp

SPApplication *app = SPApplication::GetInstance(); // przykład pobierania
                                                   // obiektu
```

Powyższy fragment przedstawia schemat budowy klasy opartej na singeltonie oraz przykład jej użycia. Istnieje wiele przykładów wykorzystania tego wzorca, ja wybrałem wersję opartą na wskaźniku. Powyższy kod jest przykładem leniwej inicjalizacji. Podczas uruchomienia programu, obiekt *SPApplication* nie istnieje, dopiero podczas pierwszego wywołania funkcji *GetInstance* obiekt zostaje utworzony. Samo tworzenie obiektu powinno odbywać się zgodnie z zasadami dostępu do sekcji krytycznej, w celu uchronienia się przez stworzeniem wielu obiektów w innych wątkach. Ostatnim ważnym wymaganiem wzorca jest ukrycie konstruktorów oraz wskaźnika na obiekt, uniemożliwiając dokonanie niechcianych operacji:

```
// main.cpp
SPApplication *app1 = new SPApplication (); //BŁĄD!!! nie można tworzyć obiektów
SPApplication app2 = *( EngineCore::GetInstance() );  // BŁĄD!!! nie można
                                                      // kopiować
```

Znakomita większość gier otwiera okno bądź więcej okien w których odbywa się prezentacja przebiegu gry i komunikacja z graczem. Wyjątkiem od tej reguły są gry przeglądarkowe, które wykorzystują okno aplikacji przeglądarki oraz komunikację przez protokół sieciowy. Z racji ścisłego powiązania głównego okna aplikacji i samej gry, to właśnie rdzeń silnika przechowuje referencje okna aplikacji oraz kontekst graficzny OpenGL. Parametry gry takie jak szerokość i wysokość okna są publicznie dostępne do odczytu przez obiekty posiadające dostęp do obiektu rdzenia, w szczególności przez część kodu odpowiedzialną za renderowanie grafiki. Obecna implementacja jest przygotowana pod platformę *Windows XP* lub nowsze, dlatego też rdzeń jest silnie uzależniony od produktu Microsoftu. Problem zależności od platformy od zawsze spędzał sen z powiek programistom. Różnice standardów dotyczyły API systemu operacyjnego, ale również systemów plików,

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

aż po kolejność bitów w różnych architekturach procesorów. Sposoby rozwiązywania tych problemów często były motorem tworzenia nowych rozwiązań, od bibliotek w językach natywnych udostępniających wspólny interfejs ale implementacje zależne od danej platformy, aż po całe środowiska uruchomieniowe i maszyny wirtualne wykonywujące kod w językach pośrednich, jak ma to miejsce w języku Java, C# czy ActionScritpt. Niezależność od platformy można implementować na trzech poziomach:

- Środowiska (Java RE, .Net, Flash Player)
- Biblioteki (QT, Allegro, SDL, boost)
- Kodu (dyrektywy preprocesora)

W zaimplementowanych silniku wywołania API systemu ograniczone są do tworzenia okna, operacji na muteksach, pobierania wartości czasomierza (ang. timer). Te stosunkowo niewielkie uwikłanie kodu w wybraną platformę zostało rozwiązane na poziomie kodu. Poniższy przykład przedstawia wykorzystanie dyrektyw preprocesora do uniezależnienia kodu:

```
#ifdef _WIN32
	#define _MUTEX_UNLOCK( X )  ( ReleaseMutex( (X) ) )
#elif __linux__
	#define _MUTEX_UNLOCK( X ) ( pthread_mutex_unlock ( (X) ) )
#else
	#error Nieznany system operacyjny
#endif

//main.cpp
_MUTEX_UNLOCK(Mutex1);
```

Powyższy kod został obudowany klasami udostępniającymi interfejs do składowych systemu operacyjnego. Pełną listę definicji preprocesora poszczególnych systemów operacyjnych można znaleźć tutaj [25].

Kolejną rolą rdzenia silnika jest odbiór komunikatów od użytkownika oraz propagacje zdarzeń do obiektów. Zgodnie z standardem win api zdarzenia są przetwarzane w wywołaniu zwrotnym (and. callback) *WndProc* [26]. Zdarzenia generowane przez użytkownika zostały podzielone ze względu na sposób przetwarzania. Istnieją zdarzenia skierowane wprost do obiektu jak wskazanie kursorem myszy, oraz skierowane do wielu obiektów jednocześnie, jak zdarzenia wciśnięcia przez użytkownika klawisza klawiatury. W silniku oprócz wymienionych zdarzeń użytkownika, istnieje szereg zdarzeń związanych z cyklem życia obiektów jak tworzenie, zwalnianie obiektów, ładowanie zasobów czy też wyświetlanie nowej klatki animacji.

Kluczowym zadaniem rdzenia silnika jest implementacja pętli głównej gry, w przypadku tego rozwiązania jest to funkcja *Run*. Na pętlę gry składa się kilka prostych zadań wykonywanych cyklicznie:

- Odświeżenie zegara
- Mapowanie modeli wczytanych asynchronicznie
- Stworzenie zdarzenia wyświetlania klatki
- Pobranie stanu kontrolerów gry i stworzenie stosownych zdarzeń
- Wykonanie kodu użytkownika i zdarzeń
- Wygenerowanie i wyświetlenie klatki animacji
- Zmiana buforów

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

- Obliczenie długości trwania przebiegu pętli

Na chwilę uwagi zasługuje model asynchronicznego wykonywania długich operacji, w tym wypadku wczytywaniu plików z dysku, bądź sieci. Użytkownik który zdecydował się załadować nowy model trójwymiarowy bądź nowe tekstury w kodzie obiektu ma możliwość użycia funkcji:

```
bool asynchronicznie = true;
proxy->LoadResource("liscie.bmp", drzewo, asynchronicznie);
```

Funkcja odpytuje moduł Proxy czy dany plik został już wczytany i czy istnieje w zbiorze wczytanych zasobów. Funkcja nie jest blokująca, kończy się natychmiast po wywołaniu. Jeśli model istnieje w następnej klatce obiekt pojawi się w scenie gry, jeśli jednak nie ma danego modelu zostanie dodane zadanie wczytania pliku. Podczas operacji otwierania pliku, parsowania i tworzenia modelu, główna pętla gry będzie wykonywana się bez opóźnień ponieważ zadania wczytania plików są pobierane i wykonywane asynchronicznie przez drugi wątek. Po zakończeniu operacji, wczytane zasoby zostaną przypisane, a obiekt pojawi się na ekranie. Taki zabieg jest niezbędny aby długie operacje dyskowe nie zatrzymywały pętli głównej gry, jednak powoduje pewne problemy. Kopiowanie tekstur do pamięci karty graficznej musi odbywać się w wątku posiadającym aktywny kontekst graficzny OpenGL. Dla uproszczenia implementacji i zgodnie z zaleceniami [27] wywołanie funkcji kopiującej teksturę do pamięci karty:

```
glGenTextures( GLsizei n,  GLuint * textures )
```

odbywa się w kontekście wątku głównego.

Synchronizacja dwóch wątków odbywa się przy użyciu Monitora [13] oraz obiektu *SPMutex* dostępnego jako publiczne pole klasy *SPApplication*. Wspomniany semafor binarny może być używany do synchronizacji wszystkich operacji asynchronicznych ingerujących w silnik, gdyż główna pętla gry dba o jego poprawne blokowanie, podczas kluczowych momentów obiegu pętli. Na potrzeby pracy zaimplantowana została klasa szablonowa *SPMonitor*, zapewniająca synchronizację dostępu do klasy *std::map*. Z powodu braku zmiennych warunkowych [13] w systemie Windows XP [29] klasa została zaimplementowana przy użyciu semaforów [13].

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Zarządca obiektów

Ważnym zadaniem silnika gry jest udostępnienie możliwości komunikacji między obiektami gry. W tym celu został zaimplementowany moduł zarządzania obiektami, który przechowuje wskaźniki na wszystkie obiekty sceny. Dodawanie elementu do sceny odbywa się przy użyciu Fabryki [3] SPScene::*AddElementToScene* . Szczegółowy opis metody dodawania obiektów został opisany w dalszej części pracy. Główną klasą modułu zarządzania obiektów jest klasa *SPElementsIdManager*. Jak wskazuje nazwa tej klasy, jej zadaniem jest przydzielanie obiektom sceny identyfikatorów. Rolą obiektu zarządzającego elementami gry jest zachowanie unikalności nazw obiektów które są ustalane przez programistę. Z racji możliwych stosunkowo dużych liczb obiektów w scenie wskaźniki są przechowywane w strukturze *std::map*. Wymieniona struktura w ogólnym przypadku umożliwia bezpośredni dostęp do obiektów, nie wymagający przeglądania całej struktury. W sytuacji gdy użytkownik stara się dodać element nie posiadający unikalnej nazwy funkcja dodawania elementów zwróci *NULL*, obiekt zostanie zwolniony. Zwalnianie pamięci jest zalecane, gdyż elementy nie istniejące w drzewie relacji rodzic-dziecko sceny, nie wpływają na przebieg rozgrywki, ich kod nie jest wykonywany. Co więcej zwalnianie obiektu który nie został dodany jest wymagane aby zapobiec wyciekom pamięci:

```
Obj *obj = static_cast<Obj*>(Scena->AddElementToScene(„nie-unikalnaNazwa”,
                                                        new Obj() ));
if(obj == NULL)
{
        // funkcja zwróciła NULL
        // nie mamy żadnego wskaźnika na stworzony obiekt o „nie-unikalnejNazwie”
        // pamięć bezpowrotnie wyciekła z programu
}
```

Zarządca obiektów jest dostępny jako klasa bazowa dla obiektu *SPScene*. Obiekty gry nie muszą być dodawane do sceny, a komunikacja między obiektami może zostać zapewniona przez przekazywanie wskaźnika. Poniższy kod przedstawia sposób dodania elementu do sceny.

```
SPScene scena = new SPScene();
scena->AddElementToScene("bron1",new Bron());
scena->AddElementToScene("postac",new Gracz());

void Gracz::OnStart( )
{
        Bron * bron = static_cast< Bron *> ( GetScene()->\
                    GetElementById("bron1") );
        if(bron != NULL)
        {
                bron->MoveElement( this->GetPos() );
                bron->Przeladuj();
        }
};
```

Korzystając z funkcji wyszukiwania obiektów programista może bezpośrednio wpływać na przemieszczanie wyszukanego obiektu, obroty, skalowanie oraz wykonywać publiczne funkcje, jak na powyższym przykładzie. Programista musi jednak zadbać o poprawne rzutowanie obiektów, aby uchronić się przez błędami:

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

```
class Ksiezyc : public SPSceneElement;
class Ziemia : public SPSceneElement;

Ksiezyc *k = static_cast<Ksiezyc*>( Kosmos-> AddElementToScene( „ksiezyc”, new
                                                                Ksiezyc() ) );
Ziemia *ziemia = static_cast<Ziemia*> ( GetScene()->\
                    GetElementById("ksiezyc") );
if(ziemia != NULL)
{
       wyswietlNaEkran(ziemia->obwod);      // Wartość niewłaściwa
       wyswietlNaEkran(ziemia->Position.x); // Wartość równa k->Position.x
                                 // ponieważ jest to pole klasy SceneElement
}
```

Powyższy kod skompiluje się, i wykona bez błędu ponieważ istnieje rzutowanie typów *Księżyc* na *SPSceneElement* oraz *SPSceneElement* na *Ziemia*, jednak wartości pól których obiekt *Ziemia* nie dziedziczy z klasy *SPSceneElement* będą niewłaściwe.
Wyrażenie rzutowania typów

```
static_cast<X>( y )
```

jest prawidłowe, gdy jedno z wyrażeń:

```
Y y = x; lub X x = y;
```

jest prawidłowe. Wynikający z tego problem rozwiązuje standard języka C++ przez wprowadzenia rzutowania dynamicznego. Jest ono podobne do rzutowania statycznego jednakże dokonywane jest sprawdzenie hierarchii dziedziczenia klas. Rzutowanie dynamiczne w przypadku niemożliwej do wykonania konwersji na referencjach wyrzuca wyjątek *std::bad_cast*, natomiast w przypadku wskaźników ustawia zmienną na NULL [12].
W powyższym przykładzie rzutowanie statyczne powinno zostać zmienione na *dynamic_cast*, i w wyniku wykonywania kodu do wskaźnika zostanie przypisana wartość 0, podobnie jak w sytuacji w której funkcja *GetElementById* nie znalazła obiektu:

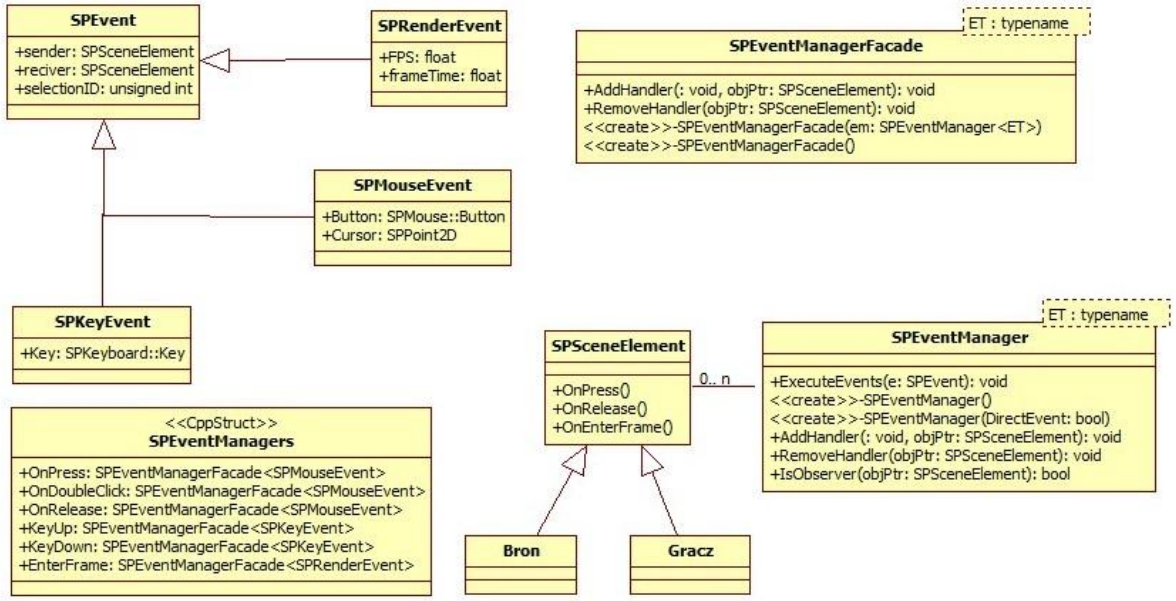
```
ziemia = dynamic_cast<Ziemia*> ( GetScene()->\
                    GetElementById("ksiezyc") );
if(ziemia != NULL){ /* operacje na wskaźniku ‘ziemia’ */ };
else{ /* Logowanie błędu – obiekt nie znaleziony lub nieprawidłowe rzutowanie */
};
```

Ostatnią funkcją jaka implementuje klasa jest usuwanie obiektów ze sceny. Wywołanie funkcji *RemoveElement(std::string id)* spowoduje usunięcie elementu z struktury map, oraz usunięcie z listy obserwatorów zdarzeń użytkownika. Jeśli obiekt posiadał przypisany model trójwymiarowy, tekstury, lub inne zasoby, nie zostaną one zwolnione, gdyż mogą być używane przez inne elementy sceny. Zwolnienie zasobów, jeśli jest konieczne, powinno odbyć się odpowiednią funkcją modułu zarządzania zasobami.

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Zarządca zdarzeń

Implementacja modułu zarządzania zdarzeniami została umieszczona w plikach *SPEvent*. Poniższy diagram przedstawia klasy modułu:



Rys. 2 Diagram klas modułu Zarządca zdarzeń

Założeniem projektowym było zapewnienie aplikacji własnej kolejki zdarzeń, opartej na natywnym API systemu operacyjnego umożliwiającą minimalizację wywołań funkcji obsługi zdarzeń przez realizację postulatu „domyślnie odłączony”. Jednocześnie moduł zarządzania zdarzeniami powinien udostępniać użytkownikowi łatwy interfejs. Założenia projektowe można przedstawić kilku punktach:

- Minimalizacja kodu
- Hermetyzacja klas – zamiast statycznej funkcji obsługi wewnątrz klasy
- Obsługa zdarzeń domyślnie wyłączona
- Budowa na bazie API systemu
- Wygodny interfejs wiązania zdarzeń i obsługi
- Dwa typy zdarzeń – bezpośredni lub rozgłaszanie
- Dynamiczne wiązanie i rozwiązywanie połączeń Zdarzenie – Obsługa
- Wykorzystanie mechanizmu polimorfizmu

Założenie minimalizacji kodu z punktu widzenia użytkownika nie jest ważnym wymaganiem, gdyż ma on dostęp do skompilowanej biblioteki. Jednakże sam etap projektowania, oraz późniejsze zmiany jakie będą wprowadzane w silniku muszą być trywialne. Niedopuszczalne jest mnożenie klas różniących się jedną linią kodu, a mianowicie wywołującą funkcję obsługi zdarzenia. Poniższy przykład obrazuje ten błąd projektowania:

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

```
class OnPressExecutor
{
	/*  pola klasy */
	void ExecuteEvent(MouseArgs e)
	{
		e.Reciver->OnPress(e);
	}
};
class OnReleaseExecutor
{
	/*  te same pola klasy */
	void ExecuteEvent(MouseArgs e)
	{
		e.Reciver->OnRelease (e);
	}
};
```

Powyższy problem w sposób naturalny rozwiązuje polimorfizm, poprzez wydzielenie klasy *Executor* zawierającej wirtualną funkcję *ExecuteEvent*. Tworzenie obsługi nowego zdarzenia odbywać będzie się przez przeciążanie jednej funkcji. Ta prosta kompozycja nie jest jednak wystarczająca, ponieważ zdarzenia w systemie to nie tylko zdarzenia przesuwania, naciskania i zwalniania klawiszy myszki. Pomysł dziedziczenia interfejsu nadal nie jest doskonały:

```
class EventExecutor
{
	void virtual ExecuteEvent(MouseArgs e);
		// problem z przekazaniem argumentów
		//wywołania innych niż MouseArgs
}
```

Ten problem, brutalnie mógłby być rozwiązany poprzez rozmnożenie klas na zdarzenia klawiatury i myszki, lecz dużo lepszym rozwiązaniem wydaje się znalezienie sposobu na takie zmodyfikowanie argumentów aby funkcja była bardziej uniwersalna. Przykładami takich rozwiązań programów pisanych w języku C są unie lub przekazywanie parametrów jako wartość binarna jak to ma miejsce w funkcji WinProc w argumentach lparam i rparam. Karkołomnym rozwiązaniem może być również przekazywanie wskaźnika na void, i wiara iż programista nie pomyli się w wywołaniu funkcji i dokona poprawnego rzutowania. W standardzie języka C++ operacje rzutowania stają się o wiele bardziej bezpieczne jednak nadal nie chroni przed błędami. Uniwersalny obiekt szablonowy powinien zdawać sobie sprawę z tego jakie argumenty funkcji powinien przyjmować a jakie odrzucać i logować błąd.

```
template<typename ET>
void EventManager<ET>::ExecuteEvents(Event *e)
{
	if(e->GetObjType() != this->Type)
	{
	return; // typ zdarzenia nieprawidłowy – koniec funkcji
	}
	foreach(obj in Objects/*każdy obiekt który czeka na dane zdarzenie*/)
	{
      ET *ev = (ET*) e;
		obj-> ExecuteEvents (ev);
	}
};
```

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Powyższe rozwiązanie nadal nie jest kompletne, analizując kod łatwo zauważyć, że o ile bezpieczeństwo rzutowania zostało zapewnione, o tyle nadal istnieje konieczność mnożenia klas implementujących wywołanie poszczególnych funkcji obsługi. Mechanizmem który został wykorzystany w tej pracy do rozwiązania problemu wiązania funkcji jest wskaźnik na funkcję. Obiekt klasy zarządcy zdarzeń przechowuje wskaźnik na funkcję oraz na obiekt, dzięki temu zabiegowi ostatnia linia kodu może zostać zastąpiona wywołaniem:

```
template<typename *ET> // ET - typ zdarzenia
foreach (observer in list<std::pair<void
                    (SPSceneElement::*)(ET*),SPSceneElement*> > )
{
       ((*observer).second->*((*observer).first))(eventArgptr);
}
```

Posługiwanie się wskaźnikami na funkcję w języku C++ jest jednak problematyczne. Mechanizm został zastąpiony przez funkcję wirtualne jednak wspomniane funkcje nie dają możliwość jakie oczekujemy od systemu, powraca problem z mnożeniem klas zarządcy zdarzeń różniących się implementacją wywołania funkcji obsługi. Problemem który mocno ogranicza mechanizm jest dostawianie przez kompilator wskaźnika na obiekt przed wskaźnikiem na funkcje:

```
void (*)(Args*);           // w ten sposób nie można przekazać wskaźnika na
                           // nie statyczną funkcję klasy
void (NazwaKlasy::*)(Args *);// aby przekazać wskaźnik należy wyspecyfikować
                             // jaka to klasa.
```

Funkcje obsługi muszą być zadeklarowane jako funkcje wirtualne w klasie bazowej, a programista nie ma dowolności w tworzeniu funkcji obsługi w swojej klasie, musi nadpisywać funkcje istniejące w klasie bazowej. Samo poprawne przypisanie wskaźnika funkcji do klasy Obserwatora jest zadaniem uciążliwym. Programista musi pamiętać o znaku wyłuskania, i podania przestrzeni nazw klasy. Poniższy przykład ilustruje poprawne wiązanie funkcji i typu zdarzenia w ciele obiektu tworzonego przez użytkownika:

```
Application->EventHandlers.\
       OnPress.AddHandler(&SceneElement::OnPress, this);
```

Powyższą niezbyt wygodną operacje można zastąpić wywołaniem makra zdefiniowanym w pliku *SimpleProject.h*:

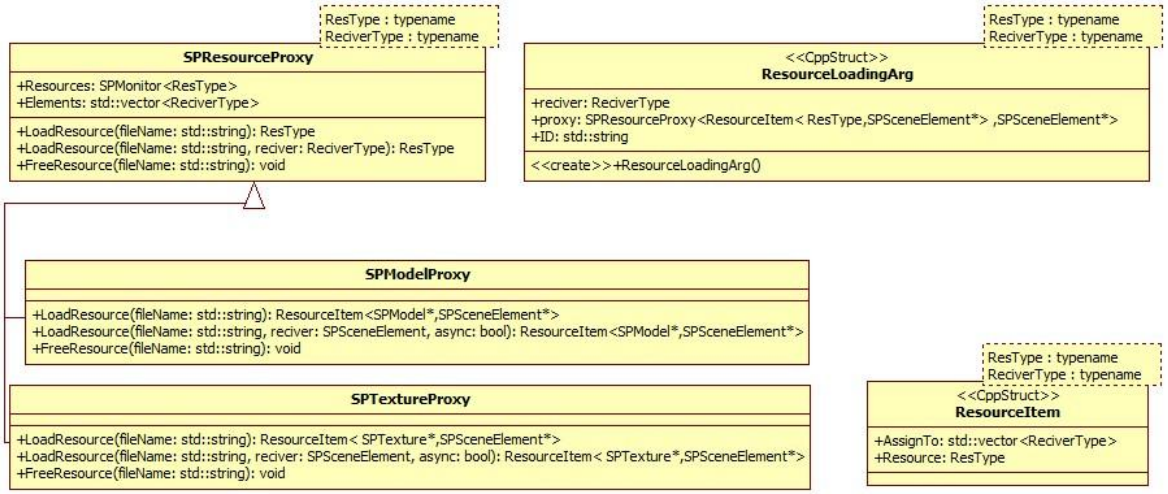
```
// zdarzenie typu KeyIsDown jest obsługiwane przez funkcję OnKeyIsDown
BindHandler(KeyIsDown,OnKeyIsDown);
```

Odpowiedzią na powyższe problemy w przekazywaniu wskaźnika na funkcje klasy jest wprowadzony w standardzie C++11 mechanizm wiązania i przekazywania funkcji [24].

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Moduł zasobów

Implementacja modułu zarządzania zasobami została umieszczona w plikach *SPResource.h, cpp* oraz *hpp*. Poniższy diagram przedstawia klasy modułu:



Rys. 3 Diagram klas modułu Zasoby

Implementacja zarządzania obiektami miała za zadanie spełniać sześć założeń związanych z redukcją alokacji pamięci oraz szybkości działania procesu przydzielania zasobów. Było to:

- Minimalizacja użycia pamięci operacyjnej przez zasoby
- Dynamiczne zwalnianie i wczytywanie zasobów w czasie gry
- Nie blokowanie długimi operacjami głównej pętli gry
- Zapewnienie interfejsu do modeli i tekstur tworzonych w zewnętrznych programach
- Wielowątkowość i synchronizacja
- Dostęp do zasobów - model Pośrednika (ang. proxy)

Zarządca zasobów jest obiektem dostępnym statycznie, jest rozszerzeniem wzorca *Singleton* [3]. Jego zadaniem jest przechowywanie i zarządzanie przydziałem wszystkich dostępnych w grze zasobów, jak pliki dźwiękowe, modele trójwymiarowe. W obrębie jednej aplikacji, częstym zjawiskiem, może być wielokrotne wczytywanie tych samych zasobów. Zauważenie tego faktu zachęciło autora do poszukania rozwiązań, zapewniających ograniczenie zużycia pamięci operacyjnej przez silnik, oraz lepsze zarządzanie zasobami. Ciekawym oraz dobrze znanym rozwiązaniem problemu zarządzania zasobami okazał się wzorzec Pośrednika [3] (ang. proxy). Podczas operacji ładowania zasobów, moduł zarządcy dokonuje sprawdzenia na zlecenie klienta, którym najczęściej jest kod użytkownika, czy dany element istnieje w pamięci operacyjnej. Na podstawie tego sprawdzenia dokonywany jest wybór, czy pobierać zasób z dysku, czy operacja ta jest niepotrzebna. Taki algorytm dostępu do zasobów uniemożliwia zapełnianie pamięci tą samą zawartością, oraz marnowanie czasu procesora. Z drugiej strony użytkownik kodu powinien mieć możliwość zwalniania zasobów by nie dopuścić do całkowitego zapełnienia pamięci operacyjnej. Możliwość zwalniania pamięci zasobów na żądanie może okazać się pewną pułapką. Istnieje olbrzymie prawdopodobieństwo, iż programista nieopacznie zwolni pamięć przechowującą zasób z którego korzysta kilka obiektów powodując problemy, i spowoduje nieoczekiwane

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

zachowanie gry. W przypadku modeli trójwymiarowych wszystkie obiekty przestaną być wyświetlane, lecz ich kod nadal będzie wykonywany. Niewielkim nakładem pracy mechanizm zwalniania zasobów można usprawnić i ulepszyć wzorując się na rozwiązaniach zwalniania pamięci z języków Java czy C# GC, czy też inteligentnych wskaźników z biblioteki boost czy standardu C++11. Każdy zasób jaki jest pobierany do pamięci operacyjnej został powiększony o listę wskaźników na obiekty które używają danego zasobu. Podczas niszczenia obiektów silnik automatycznie usuwa wskaźnik współwłaściciela zasobu. Operacja usuwania obiektu lub zmiany zasobu została rozszerzona o sprawdzenie czy dany zasób ma właścicieli. W przypadku braku właścicieli pamięć operacyjna lub karty graficznej, jest zwalniana. Przedstawiony mechanizm prezentuje się znacznie lepiej, jednak zautomatyzowanie usuwania zasobów może nieść za sobą problemy. Rozpatrzmy przykład prostej gry, spadające elementy muszą być łapane w kosz sterowany przez gracza. Gdy element wpada do kosza jest niszczony. Zabawa ponawia się po każdej turze. W powyższym przykładzie zasoby gry przypisane do spadających elementów zostaną zwolnione z pamięci gdy ostatni element wpadnie do kosza, jednakże każda kolejna fala spadających przedmiotów będzie zaczynała się ładowaniem tych samych zasobów. W zaimplementowanym silniku ten problem został rozwiązany przez udostępnienie programiście flagi wskazującej czy zasób może być zwalniany automatycznie czy nie. Domyślnie ustawiona flaga informuje silnik o automatycznym zwalnianiu wszystkich zasobów.

Kolejnym, wspomnianym wcześniej, problemem implementacyjnym jest ładowanie zasobów bez konieczności wprowadzania znacznych opóźnień renderowania grafiki. Pokaźnych rozmiarów pliki zasobów, oraz fakt wolniejszego o rzędy wielkości czasu dostępu do dysku w porównaniu z pamięcią operacyjną, implikują czasowe wstrzymanie gry na dłuższy czas. Fakt umieszczenia obsługi zasobów w wątku wykonującym pętlę gry, sprawia iż poniższy kod może zablokować całą grę na jakiś czas:

```
void Samochod::OnStart( )
{
	// gdy model jest w pamięci, wywołanie kończy
	// się natychmiast
	// gdy nie ma, wykonuje się długi czas blokując grę
	this->Model = GetModelFromFile("duzyPlik.obj");
};
```

Z pomocą w rozwiązaniu problemu przychodzi mechanizm funkcji nieblokujących. Obsługa wczytywania obiektu została rozdzielona na wątki a sama funkcja przybrała nowy kształt:

```
bool async = true;
// funkcja blokująca
// jeśli w pamięci istnieje model kończy się natychmiast
// jeśli nie, kończy się po odczytaniu pliku z dysku
samochod->DisplayModel = proxy->LoadResource("fiat.obj", 0, !async)
// funkcja nieblokująca tworzy asynchroniczne zadanie ładowania zasobu
// w parametrze przekazywany jest wskaźnik na odbiorcę zdarzenia
// załadowania zasobu
proxy->LoadResource("fiat.obj", samochod, !async)
```

Wywołanie funkcji *LoadResource* z parametrem *true* bez względu na wynik sprawdzenia czy dany zasób istnieje czy nie, kończy się natychmiast i główna pętla gry nie jest w znaczny sposób wstrzymywana. Jednocześnie zlecenie klienta trafia w zależności od wyniku sprawdzenia na dwa stosy:

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

- Zleceń oczekujących na pobranie zasobów z dysku w przypadku gdy dany zasób jest wczytywany na rzecz innego obiektu
- Zleceń do wykonania

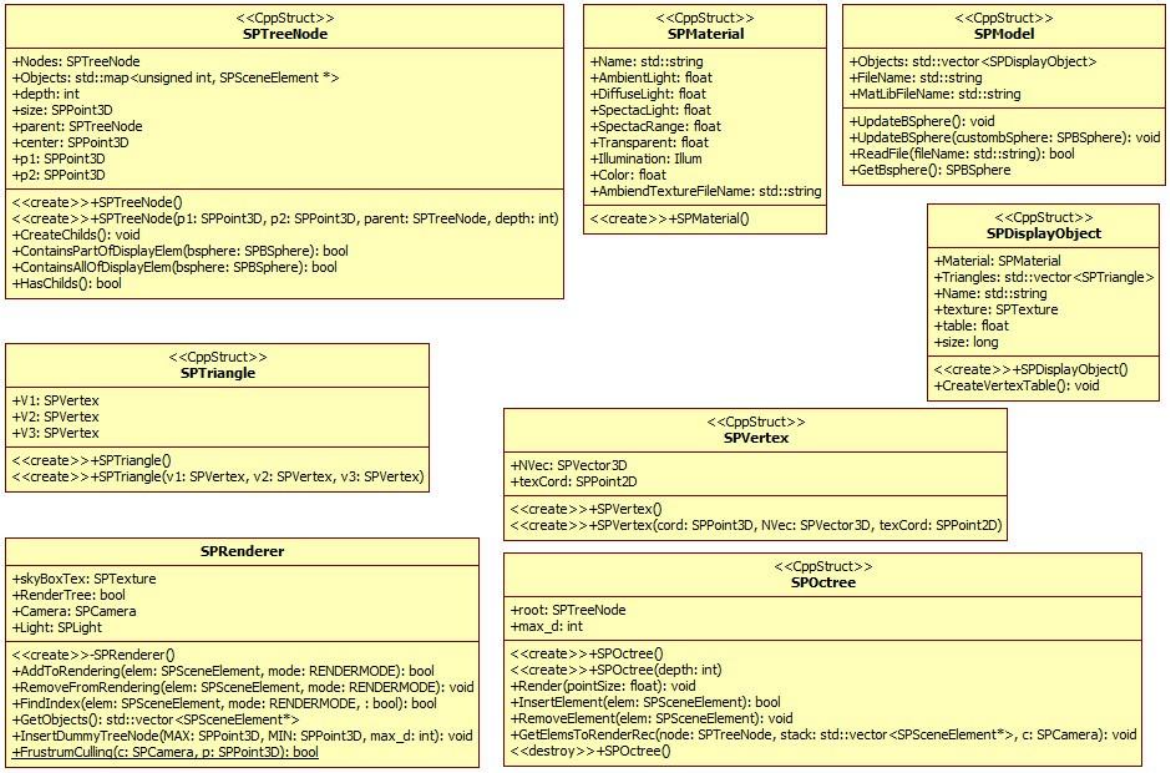
Oba wątki, główny i wczytujący zasób, są więc w relacji producent-konsument [13]. Synchronizacja tych wątków została oparta na Monitorze dostępu zaimplementowanego jako klasa szablonowa, obudowująca strukturę *std::map* synchronizowanymi funkcjami *Push* i *Pop*. Dostęp do mapy jest chroniony przez semafor z powodu braku zmiennych warunkowych w *Windows XP* [28]. Warto nadmienić, iż przypadku braku zleceń ładowania zasobów nie zostają uruchomione żadne dodatkowe wątki i nie zajmują czasu procesora. W przypadku architektury dwurdzeniowej jest to nieoptymalne wykorzystanie czasu procesora. Mała zajętość jednego z rdzeni daje możliwość umieszczenia w kodzie kolejnego wątku np. obliczeń fizyki.

Warto nadmienić, iż przypadku plików z modelami trójwymiarowymi, długotrwałą operacją jest nie tylko dostęp do dysku ale również parsowanie, oraz interpretacja zawartości pliku. Mechanizm ładowania musi jednak dbać o jak najszybsze opuszczenie sekcji krytycznej, w związku z tym wszelkie operacje na wczytywanych zasobach odbywają się w wątku wczytującym zasób z dysku, a dostęp do sekcji krytycznej ogranicza się jedynie do przepisania wskaźnika na przetworzony zasób.

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Moduł renderowania

Implementacja modułu zarządzania zdarzeniami została umieszczona w plikach *SPRenderer.h, cpp* oraz *hpp*. Poniższy diagram przedstawia klasy modułu:



Rys. 4 Diagram klas modułu Renderowanie

Moduł renderowania udostępnia szereg klas umożliwiających wyświetlanie obiektów na ekranie. Główną klasą modułu, *Singletonem*, jest klasa *SPRenderer*. Obiekt tej klasy jest odpowiedzialny za renderowanie obiektów opisanych klasami *SPModel* i *SPMaterial*. Obiekty wyświetlane są przechowywane w trzech kolejkach; zawsze wyświetlane, przechowywane w drzewie ósemkowym, wyświetlane we współrzędnych kamery. Moduł definiuje algorytmy usuwania niewidocznych obiektów (ang. viewing frustum culling), jeśli korzystamy z przechowywania obiektów w drzewie ósemkowym.

Kolejnym zadaniem zarządcy jest nadawanie dodanym obiektom unikalnego identyfikatora selekcji. Jest on niezbędny do tworzenia stosu nazw obiektów renderowanych za pomocą funkcji

```
glRenderMode( GL_SELECT );
```

Programista nie posiada bezpośredniego wpływu na tworzony identyfikator. Obiekt klasy *SPRenderer* przypisuje identyfikator elementom sceny, będący liczbą dodatnią o precyzji 64 bitowej, oraz inkrementuje licznik. Mechanizm renderowania w trybie selekcji jest uruchamiany podczas wciśnięcia klawisza myszy w głównym oknie gry. Wciśnięcie przycisku myszy prócz standardowego zdarzenia wywołuje funkcje selekcji, która zwraca identyfikator selekcji obiektu znajdującego się pod kursorem myszy. Aby zoptymalizować długość wywołania funkcji *glRenderMode(GL_SELECT)* renderowane są jedynie obiekty które zostały zarejestrowane jako odbiorcy zdarzenia *OnPress* lub *OnRelease*, przy pomocy makra *BindHandler* lub funkcji:

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

```
app->Events.OnPress(&SPSceneElement::OnPress,obiekt);
```

Dodatkowo pomijane jest teksturowanie oraz światła.

Ciało funkcji renderowania w trybie selekcji przedstawiłem poniżej:

```
unsigned int SPMainWindow::GetSelectedItem(int x, int y){
	SPRenderer * renderer = SPRenderer::GetInstance();
	int BUFF_SIZE = 64; // w różnych wersjach bibliotek wartość może
			// być większa. 64 to wartość minimalna
	unsigned int min_vertex = 0;
	unsigned int selID = 0;
	GLuint select_buffer[ BUFF_SIZE ];
	glSelectBuffer( BUFF_SIZE, select_buffer );
	int viewport[ 4 ];
	glGetIntegerv( GL_VIEWPORT, viewport );
	int width = viewport[ 2 ];
	int height = viewport[ 3 ];
	glMatrixMode( GL_PROJECTION );
	glPushMatrix();
	glLoadIdentity();
	// ograniczenie renderowania do prostopadłościanu o boku 2x2 piksele
	gluPickMatrix( x, (height - y ) , 2, 2, viewport );
	bool toSelect = true;
	renderer->Resize(toSelect);
	glRenderMode( GL_SELECT );
	glInitNames();
	glPushName( 0 ); // inicjujemy stos
	glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
	glMatrixMode( GL_MODELVIEW );
	glDisable(GL_LIGHTING);
	glEnable( GL_DEPTH_TEST );
	glDepthFunc( GL_LESS );
	glPushMatrix();
	glLoadIdentity();
	renderer->Camera->TransformScene();
	renderer->Render(toSelect);
	glPopMatrix();
	GLint hits = glRenderMode( GL_RENDER );
	glMatrixMode( GL_PROJECTION );
	glPopMatrix();
	if(hits > 0){ // trafiono co najmniej jeden obiekt
		if( select_buffer[ 0 ] == 1 ){ // pierwszy obiekt
			min_vertex = select_buffer[ 1 ]; // piksel obiektu
							// najbliżej obserwatora
			selID = select_buffer[ 3 ];	// id selekcji
		}
		for(int i = 0; i < hits; i ++){ // kolejne trafienia jeśli są
			if( select_buffer[ i * 4 ] == 1 ){
				if(min_vertex > select_buffer[ i * 4 + 1 ] ){
					min_vertex = select_buffer[ i * 4 + 1 ];
				selID = select_buffer[ i * 4 + 3 ];
				}
			}
		}
		return selID;
	}
	return 0;
}
```

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Na początku funkcji tworzy się stos nazw obiektów. Wielkość stosu jest podyktowana implementacją i wersją biblioteki, jednak nie jest mniejsza niż 64. Niezależnie od wersji biblioteki istnieje możliwość podejrzenia jaką wielkość może mieć stos nazw poprzez użycie funkcji *glGetIntegerv* [35]:

```
int max = -1;
// funkcja zwraca tylko jedna wartość
glGetIntegerv(GL_MAX_NAME_STACK_DEPTH, &max);
```

Kolejnym wykonywanym krokiem jest pobranie parametrów widoku (ang. viewport) w celu poprawnego ustawienia bryły obcinania oraz odłożenie na stos nowej macierzy projekcji. Korzystając z funkcji biblioteki GLU bryła obcinania zmniejszona jest do sześcianu w miejscu kursora o podstawie 2x2 piksela. Bez wywołania funkcji gluPickMatrix w buforze selekcji znalazłyby się wszystkie elementy widoczne na ekranie.
Cała scena jest renderowana w trybie selekcji korzystając z przedstawionej funkcji, a następnie podczas zmiany trybu renderowania zostaje zwrócona wartość trafień (ang. hits) informująca ile obiektów zostało wyrenderowanych pod kursorem myszy. Kolejnym krokiem jest pozbycie się ze stosu tymczasowej macierzy projekcji stworzonej na potrzemy selekcji obiektów i przetwarzanie bufora selekcji.
Biblioteka OpenGL umożliwia tworzenie hierarchiczne stosy nazw, poniższy przykład ilustruje sposób tworzenia grup nazw obiektów:

```
// obiekt Kolo_Lewe z grupy Samochod
glLoadName( samochod );
glPushName( Kolo_Lewe );
glPopName();

// obiekt Kolo_Prawe z grupy Samochod
glPushName( Kolo_Lewe );
glPopName();

// obiekt Drzewo nie grupowany
glLoadName( Drzewo );
```

W zależności od wartości zmiennych trafień, oraz wartości zapisanych w buforze stos obiektów będzie wyglądał inaczej. Przykładowo jeśli w wyniku selekcji zostanie wybrane lewe koło samochodu to zmienne przybiorą następujące wartości:

```
hits == 1 // jeden obiekt został wybrany
select_buffer[0] == 2 // na stosie są dwie nazwy
select_buffer[3] == Samochod
select_buffer[4] == Kolo_Lewe
```

Natomiast w przypadku wyboru drzewa:

```
hits == 1 // jeden obiekt został wybrany
select_buffer[0] == 1 // na stosie jest jedna nazwa
select_buffer[3] == Drzewo
```

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Szczegóły implementacji i działania selekcji obiektów można znaleźć tutaj [36].
Kolejną funkcjonalnością klasy *SPRenderer* jest włączenie mieszania pikseli buforów ramki przy pomocy funkcji *glEnable(GL_BLEND)*. Piksele sceny zostają zmieszane zgodnie z ustaloną funkcją z pikselami obiektów które mają włączoną opcję mieszania. Poniższy przykład przedstawia sposób włączenia mieszania.

```
//Maska : public SPSceneElement
Maska maska = new Maska();
maska->blend = true;
```

Obiekty z ustawioną flagą mieszania, *blend,* zostają renderowane wraz ze sceną. Mieszanie pikseli odbywa się zgodnie z funkcją *glBlendFunc(GL_ONE_MINUS_SRC_COLOR,GL_SRC_COLOR)*. Mechanizm jest przystosowany do tekstur nie posiadających kanału alfa. Współczynnikiem mieszania jest skala szarości. Stosowana maska całkowicie ukrywa piksele czarnym kolorem, oraz jest całkowicie przeźroczysta w miejscach koloru białego. Zgodnie z wyżej wymienioną funkcją:

$$piksel_{wynikowy} = (1 - Kolor_{maski}) * Kolor_{maski} + (Kolor_{maski}) * Kolor_{sceny}$$

Rys. 5 Maska, tło oraz wynik mieszania

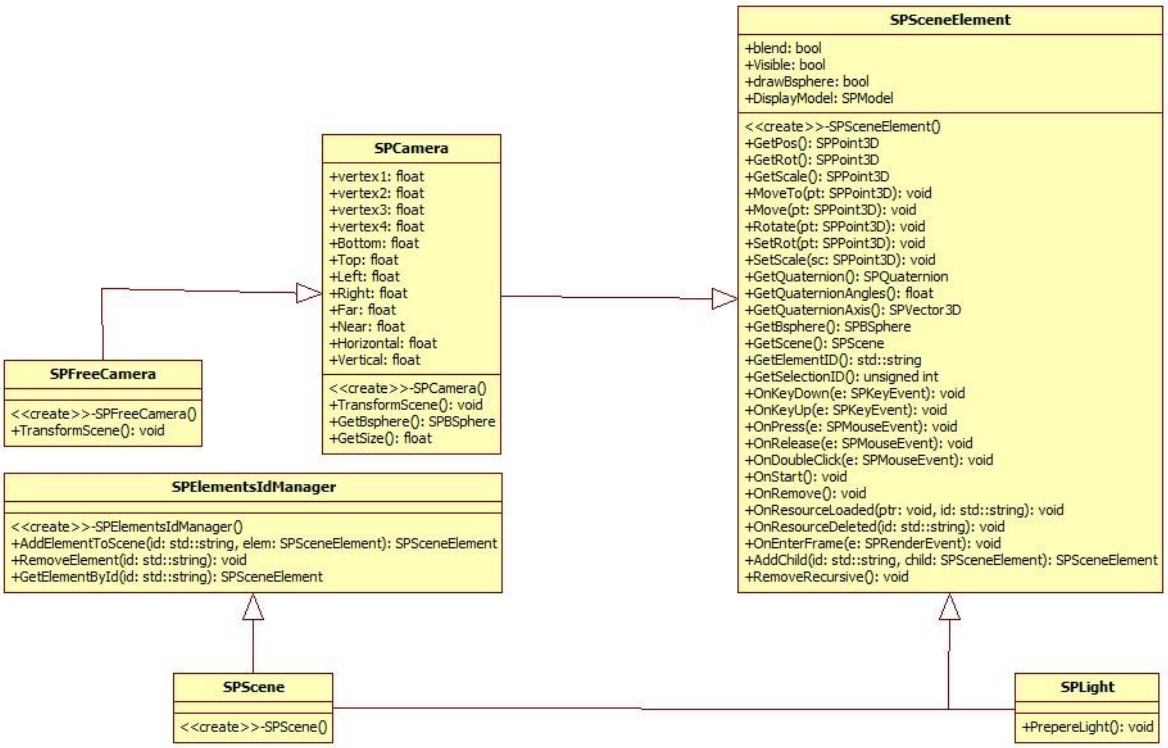
Moduł renderowania udostępnia również prosty interfejs do tworzenia *Sky Box*. Po przypisaniu obiektu *SPTexture* do jednej ze zmiennych klasy *SPRenderer* wokół kamery zostanie wygenerowana oteksturowana sfera. Jej wielkość jest proporcjonalna do wielkości tekstury, jednak nie większa niż granica widoczności. Aby wyeliminować problemy z przecięciami sfery nieba z obiektami, *Sky Box* renderowany jest z wyłączonym buforem głębi (ang. z-buffer). Poniższy przykład przedstawia przypisanie tekstury i włączenie renderowania nieba.

```
renderer->skyBoxTex = niebo;      // przypisanie wskaźnika na obiekt SPTexture
```

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Moduł sceny

Implementacja modułu sceny została umieszczona w plikach *SPScene.h*, *cpp* oraz *hpp*. Poniższy diagram przedstawia klasy modułu:



Rys. 6 Diagram klas modułu Scena

Moduł sceny łączy kilka klas opisujących scenę trójwymiarową. W szczególności jest to *SPSceneElement* będąca bazą do wszystkich tworzonych przez użytkownika obiektów, wyświetlanych lub istniejących w scenie jak światła czy kamery, oraz samej sceny trójwymiarowej *SPScene*. Połączenia wszystkich elementów sceny w strukturę drzewiastą, hierarchie rodzic-dziecko jest zaimplementowane dzięki posiadaniu przez wszystkie wymienione klasy wspólnej bazy *SPSceneElement*.

Klasa *SPSceneElement* jest reprezentantem obiektów wyświetlanych w scenie. Posiada pola opisujące jej położenie, obrót oraz skalowanie oraz zestaw funkcji do edycji tych atrybutów. Każdy element napisany przez użytkownika musi dziedziczyć po klasie *SPSceneElement* aby mógł zostać dołączony do sceny, wyświetlany oraz mógł nasłuchiwać zdarzeń i je obsługiwać. Obsługa zdarzeń jest implementowana przez poniższe funkcje wirtualne:

```
virtual void OnKeyDown     ( KeyEvent *event);
virtual void OnKeyIsDown   ( KeyEvent *event);
virtual void OnKeyUp       ( KeyEvent *event);
virtual void OnPress       ( MouseEvent *event);
virtual void OnRelease     ( MouseEvent *event);
virtual void OnMouseUp     ( KeyEvent *event);
virtual void OnMouseDown   ( KeyEvent *event);
virtual void OnEnterFrame  ( LifeCycleEvent *event);
virtual void OnCustomEvent ( CustomEvent *event);
virtual void OnResourceLoaded ( void *res, std::string id );
virtual void OnResourceDeleted( void *res, std::string id );
```

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Stworzona przez użytkownika klasa powinna implementować wybrane funkcje oraz nawiązać połączenie między zdarzeniem a funkcją makrem *BindHandler*, lub w poniższy sposób:

```
SPApplication *app = SPApplication::GetInstance();
app->Events.EnterFrame.AddHandler(&SPSceneElement::OnEnterFrame,this);
```

Programiści korzystający z silnika w łatwy sposób mogą tworzyć nowe obiekty gry, przy pomocy rozszerzania klasy *SPSceneElement*, poniższy przykład przedstawia definicję prostej gry zręcznościowej sprawdzającej refleks gracza. Po rozpoczęciu gry, na ekranie pojawi się obiekt między drugą a piątą sekundą rozgrywki. Zadaniem gracza jest jak najszybsze kliknięcie na obiekt aby znów znikł. Czas reakcji jest wyświetlany na ekranie.

```
class Refleks : SPSceneElement
{
float opoznienie;
float reakcja;
	void NowaGra()
	{
		opoznienie = rand()%3000 + 2000;
		Visible = false;

	}
public:
	Refleks()
	{
		NowaGra();
		SPApplication *app = SPApplication::GetInstance();
		app->Events.EnterFrame.AddHandler
				(&SPSceneElement::OnEnterFrame,this);
		app->Events.OnPress.AddHandler(&SPSceneElement:: OnPress,this);

	}
	// ta funkcja uruchamia się w każdej klatce animacji
	void OnEnterFrame(SPRenderEvent * e)
	{
		float frame = e->frameTime / 1000.0f;
		opoznienie -= frame;
		if(opoznienie < 0)
		{
			Visible = true;
			reakcja += frame;
		}
	}
	// na funkcja wykona się po kliknięciu myszą na obiekt
	void OnPress(SPMouseEvent *e)
	{
		if(Visible)
		{
			Wyswietlacz * w = this->GetScene()->GetElementById("w1");
			w->PokazWynik(reakcja);
			NowaGra();
		}
	}
};
```

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Moduł logowania błędów

Moduły logowania błędów oraz profilowania są ważne nie tylko z punktu widzenia produkcji gry ale również eksploatacji przez użytkowników. W dużych projektach komercyjnych, szczególnie płatnych gier MMORPG, zdiagnozowanie problemu i naprawa poprzez *łatkę* (ang. update) jest ważnym elementem świadczącym o jakości systemu. W przypadku gry World Of Warcraft amerykańskiego studia Blizzard Entertaiment, odbiorcami płatnej usługi, grania na oficjalnych serwerach, jest 7,7 miliona osób [37]. System logowania błędów i generowania raportów powinien być integralną częścią gry i po wystąpieniu błędu krytycznego, który spowodował zabicie aplikacji użytkownik gry powinien mieć możliwość wysłania szczegółowego raportu do producenta. Oprócz samych logów, informacyjnych jak i o błędach, przydatne mogą okazać się pliku zrzutu pamięci aplikacji. W systemach Windows pliki *Dump* zawierają zrzut pamięci wraz z stertą procesu. Przy pomocy tych plików oraz kodów źródłowych producenci gry mogą dokonać *odpluskwiania* (ang. debug) kodu. Jednym ze środowisk umożliwiającym odpluskwianie kodu z użyciem plików zrzutu pamięci jest Microsoft Visual Studio [38].

Zapisywanie i przechowywanie logów informacyjnych oraz o błędach może być zrealizowane na kilka sposobów. Jednym z podejść zależnych od systemu operacyjnego jest logowanie do specjalnego dziennika zdarzeń aplikacji który jest częścią systemu. Takie rozwiązanie nie jest wygodnym z punktu widzenia producenta gier ponieważ wymaga odpowiedniej konfiguracji aplikacji. Alternatywnym rozwiązaniem, prostym w implementacji oraz działaniu, jest zapisywanie zdarzeń w pliku tekstowym, lub w formacie binarnym na dysku użytkownika. W przypadku generowania raportu błędu krytycznego aplikacja może pobrać wpisy zdarzeń i dołączyć je do raportu nie angażując użytkownika. Takie rozwiązanie zostało wykorzystane między innymi w silniku Unity3D, w którym zdarzenia aplikacji są zapisywane w odpowiednich plikach.

W zaimplementowanych rozwiązaniu klasa logująca błędy wykorzystuje plik tekstowy. Wybrane rozwiązanie jest najprostsze i całkowicie przenośne na inne platformy. Wyróżnione są trzy rodzaje logów:

- Informacyjne
- Błędy
- Ostrzeżenia

Miejsce utworzenia pliku logów zostało określone w pliku konfiguracyjnym i może być zmieniane przez użytkownika. Domyślnie jest to folder w którym przechowuje się plik wykonywalny gry. Podczas działania gry do pliku zapisywane są kolejne wiersze logów, co przy nieumiejętnym wykorzystaniu przez programistę może wiązać się z pewnym niebezpieczeństwem:

```
void Kamera::OnEnterFrame( SPRenderEvent *event )
{
	if(this->Aktywna)
	{
		std::string informacja = "Kamera" + GetElementId() +
							"jest aktywna";
		Logger.WriteLog( Logger::Info, informacja );
	}
}
```

Zamierzeniem programisty było logować informację która z wielu kamer jest aktywna w danym momencie. Jednakże umieszczanie logowania informacji w funkcji *OnEnterFrame*

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

uruchamianej w zależności od wydajności komputera 30-70 razy na sekundę szybko doprowadzi do utworzenia wielogigabajtowego pliku logów. Programiści powinni odpowiedzialnie używać narzędzia jakim jest dziennik zdarzeń. Przykładem poprawnego użycia modułu może być poniższy kod:

```
void Czolg::OnStart()
{
	try
	{
		/* inicjalizacja obiektu */
	}
	catch (std::exception& e)
	{
		// wpis błędu
		Logger.WriteLog( Logger::Err, "Czolg::OnStart" + e.what());
		// wpis ostrzeżenia
		Logger.WriteLog( Logger::Warning, "Niepowodzenie przy tworzeniu "
				+  GetElementId() + ". Obiekt nie dodany do sceny" );

		/* dalsza obsługa błędu */
		return;
	}
	// wpis informacji
	Logger.WriteLog( Logger::Info, "Utworzono obiekt" +  GetElementId() );
}
```

Wyjątek wyrzucony przez funkcję musi zostać obsłużony w bloku *catch* przyjmującym w argumencie odpowiedni typ wyjątku. Jeśli wyjątek nie natrafi na odpowiedni blok *catch*, jest propagowany wyżej w hierarchii wywołań. Jeśli nie zostanie obsłużony w funkcji głównej spowoduje zabicie aplikacji. Bloki *catch* mogą być powtarzane:

```
	try{
		/* Funkcje które mogą wyrzucić wyjątek */
	} catch (const std::exception& ex) {
		/* tutaj następuje obsługa throw(std::exception) */
	} catch (const std::string& ex) {
		/* tutaj następuje obsługa throw(std::string) */
	} catch (...) {
		/* tutaj następuje obsługa reszty wyjątków */
	}
```

Obsługę wyjątków można również ustanowić poprzez funkcję biblioteki C++ w argumencie której podawany jest wskaźnik na funkcje obsługi wyjątku:

```
	void Obsluga () {
		Logger.WriteLog(Logger::Err, "Nieoczekiwane zabicie aplikacji");
		/* dalsza obsługa */
		abort();  // zabicie aplikacji
	}

	int main () {
		std::set_terminate (Obsluga);
		GenerujWyjatek();
		return 0;
	}
```

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

W przeciwieństwie do języków Java lub C# w bloku *try-catch* nie jest możliwe obsługiwanie wszystkich wyjątków. Przykładem jest prawdopodobnie najczęściej występujący – błąd dostępu do pamięci. Poniższy kod pomimo umieszczenia w bloku *try-catch* zabije aplikacje:

```
try
{
	int *i = NULL;
	*i = 3; // próba zapisu wartości 3 pod adres 0x0
	int zero = 0;
	int wynik = 12/zero; // próba dzielenia przez 0
}catch( ... )
{
	// żaden z wyjątków sprzętowych nie może być odsłużony
}
```

Wyjątki dzieli się na dwie główne kategorie:

- Sprzętowe (ang. hardware)
- Aplikacji (ang. software)

Środowiska uruchomieniowe języków Java oraz C# posiadają własną abstrakcję - maszynę wirtualną, i dzięki temu jest możliwa obsługa wyjątków sprzętowych w blokach *try-catch*. W przypadku języków natywnych, do obsługi wyjątków sprzętowych należy skorzystać z API danej platformy. W systemach Linuks są to sygnały wysyłane do aplikacji, które mogą zostać obsłużone przez programistę w kodzie natywnym. Rozwiązanie dla platformy Windows zostało oparte o mechanizm *Structured Exception Handling* [39] dbający o obsługę wyjątków sprzętowych, aplikacji oraz zabicia aplikacji przez użytkownika.

```
LONG WINAPI ObslugaBleduKrytycznego(EXCEPTION_POINTERS * ExceptionInfo)
{
	StworzZrzutPamieci(ExceptionInfo);
	return EXCEPTION_EXECUTE_HANDLER;
}
int main()
{
	// powiązanie funkcji obsługi błędu
	SetUnhandledExceptionFilter(ObslugaBleduKrytycznego);
	int *i = NULL;
	*i = 3;
	return 0;
}
```

Funkcja *StwórzZrzutPamięci* wykorzystuje gotowy mechanizm *WinApi* – funkcję *MiniDumpWriteDump* [40]. Więcej informacji o tworzeniu plików *Dump* można znaleźć w źródłach [41].

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Moduł profilowania kodu

Ważnym z punktu widzenia wydajności elementem silnika jest mechanizm dynamicznej analizy kodu w trakcie jego działania. Wraz z rozbudową i powiększaniem się kodu, analiza statyczna polegająca na czytaniu nie skompilowanego kodu w celu poszukiwania miejsc nieoptymalnych staje się trudna i pracochłonna. Skuteczność takiej analizy również nie rekompensuje wielu godzin spędzonych przez programistów wyszukujących błędy w swoim kodzie. Zaimplementowany w silniku gry system analizy kodu powinien spełniać kilka kryteriów, aby mógł zostać wykorzystany jako użyteczne narzędzie programistyczne:

- Powinien wykrywać i prezentować miejsca nieoptymalne
- Powinien być prosty w użyciu
- Informacje prezentowane przez mechanizm muszą być odświeżane dynamicznie podczas rozgrywki
- Dane powinny zostać przedstawione w strukturze drzewiastej

Przed przystąpieniem do implementacji *profilera* kodu, należy zastanowić się jakie informacje będą najbardziej użyteczne oraz w jakich przypadkach użytkownicy silnika będą sięgać do tego narzędzia. Podstawowym zadaniem analizy kodu jest określenie procentowego udziału wybranych funkcji czy modułów w całym przydzielonym czasie procesora. Informacja iż „10 procent kodu zajmuje 90 procent czasu procesora" jest bardzo cenna, gdyż bez dogłębnej analizy umożliwia wytypowanie danego fragmentu kodu jako cel optymalizacji. Schemat budowy prostego *profilera* mógłby się przedstawiać w następujący sposób:

```
void Statek::OnPress(MouseEvent *event)
{
	ProfilerNode n("Statek::OnPress");
	n.Rodzic = ProfilerManager::AktywnyWezel;
	ProfilerManager::AktywnyWezel->DodajPotomka(n.NazwaWezla);
	n.Start = SystemOperacyjny::StanCzasomierza();
	/* wykonanie kodu */
	n.Czas = SystemOperacyjny::StanCzasomierza() - n.Start;
	n.Rodzic.DodajCzas(n.Czas, n.NazwaWezla);
}
```

Na powyższym przykładzie przedstawiono sposób tworzenia węzła dziecka, obliczenie czasu trwania funkcji i odświeżeniu wartości czasomierza rodzica. Warto zauważyć, iż funkcja tworząca węzeł nie tworzy nowego obiektu na stercie przy pomocy *new* lecz obiekt węzła jest kopiowany na stos funkcji. Dzięki temu zabiegowi pod koniec wykonania funkcji programista nie musi dbać o zwalnianie pamięci po utworzonym obiekcie, ale również kod ostatnich dwóch linijek może zniknąć z ciała profilowanej funkcji i zostać przeniesiony do destruktora węzła. Podobnie druga, trzecia i czwarta linijka kodu czyli przypisanie wartości czasomierza do zmiennej *Start* i ustawienie relacji rodzic-dziecko, mogą zostać przeniesione do konstruktora klasy *ProfilerNode*. Operowanie zakresem widoczności poszczególnych obiektów stwarza mechanizmowi profilującemu kolejne możliwości. Wyobraźmy sobie przypadek, iż funkcja jest rozbudowana i chcemy zbadać czas wykonania tylko jej części, lub w raporcie wyświetlić jej części jako oddzielne węzły:

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

```
void funkcja()
{
	ProfilerNode n („funkcja”);
	bardzoDlugaFunkcja();
}

void bardzoDlugaFunkcja()
{
	{ //ograniczenie zakresu węzła 1
		ProfilerNode n („dl_fun_1”);
		/* cześć 1 kodu */

	} // koniec zakresu węzła 1 – uruchamiany destruktor
	{
		ProfilerNode n („dl_fun_2”);
		/* cześć 2 kodu */

	}// koniec zakresu węzła 2 – uruchamiany destruktor
}
```

W powyższym przykładzie ciało jednej funkcji zostało podzielone zakresem widoczności na dwie części, dzięki takiej kompozycji węzeł „funkcja” ma dwa podwęzły mimo, iż znajdują się w jednej funkcji.

Ważną cechą którą należy brać pod uwagę w budowie drzewa wywołań jest możliwość rekurencyjnego wywoływania funkcji profilowanej. W przypadku rekurencyjnego wywołania funkcji „bardzoDlugaFunkcja” dziesięć razy, węzeł „funkcja” będzie miał dwadzieścia podwęzłów zamiast dwóch. Rozwiązaniem tego problemu było sprawdzenie w statycznej klasie tworzącej nowy węzeł czy obecny węzeł ma inną nazwę niż nowotworzony. W przypadku tej samej nazwy nie jest tworzony podwęzeł a jedynie zlicza się rekurencyjne wywołania funkcji:

```
ProfilerNode(std::string nazwaWezla)
{
	ProfilerManager::StworzWezel(nazwaWezla);
}

void ProfilerManager::StworzWezel(std::string nazwaWezla)
{
	if( AktywnyWezel->Nazwa != nazwaWezla)
		AktywnyWezel = AktywnyWezel->Pobierz_PodWezel(nazwaWezla);
	AktywnyWezel->Wywolaj();
}

void ProfilerNode::Wywolaj()
{
	Wywolania ++;
	if( rekursja ++ == 0)
	{
		n.Start = SystemOperacyjny::StanCzasomierza();
	}
}
```

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

W podobny sposób należy zaimplementować destruktor klasy węzła aby sprawdzał czy stos wywołań rekurencyjnych jest równy zero:

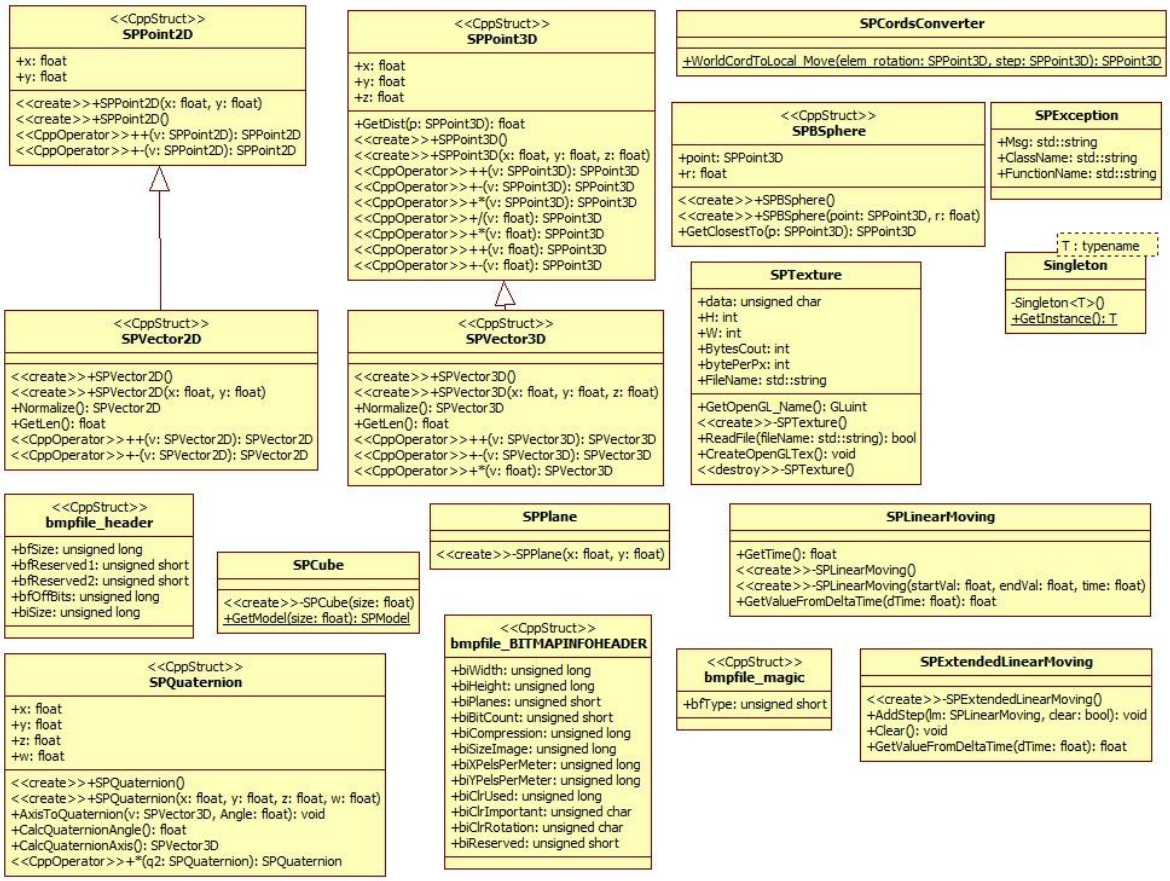
```
~ProfilerNode(std::string nazwaWezla)
{
	ProfilerManager::UsunWezel(nazwaWezla);
}

void ProfilerManager::UsunWezel(std::string nazwaWezla)
{
	if( AktywnyWezel-> Powrot())
	{
		// zapisuje konieczne informacje np. czas trwania ilość wywołań
		AktywnyWezel->Rodzic->DodajInformacjeOWezle(AktywnyWezel);
		AktywnyWezel = AktywnyWezel->Rodzic;
	}
}

bool ProfilerNode::Powrot()
{
	if(rekursja == 0 && Wywolania != 0 )
		n.Czas = SystemOperacyjny::StanCzasomierza() – n.Start;
	return (rekursja == 0);
}
```

Uzyskane przez *profiler* dane mogą być wyświetlane na ekranie komputera w czasie rzeczywistym, umożliwiając poruszanie się po strukturze drzewa. W danej chwili wybrany węzeł otrzymuje znacznik gromadzenia szczegółowych danych. Węzeł nadrzędny oraz bracia nie są przetwarzani ponieważ informacje nie muszą być prezentowane na ekranie. Informacje z węzłów podrzędnych pierwszego stopnia są gromadzone szczegółowo, między innymi informacje o procentowym udziale oraz ilości wywołań funkcji. Informacją gromadzoną z węzłów kolejnych rzędów jest czas wykonania który jest sumowany rekurencyjnie, aż do węzłów pierwszego rzędu pod wyświetlanym węzłem. Przykład implementacji został umieszczony w *Perełkach Programowania* [10].

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Moduł funkcji i klas narzędziowych

Ostatnim modułem silnika jest zestaw funkcji i klas narzędziowych. Powstawał on dynamicznie, wraz z przyrostem linii kodu silnika. Podobnie jak w projektach komercyjnych zawiera on zbiór funkcji i klas, często rozwijających możliwości istniejących klas biblioteki standardowej. Moduł został zawarty w plikach *Utils.h*, *RenderUtils.h*, *.cpp* oraz *.hpp*:



Rys. 7 Diagram klas modułu Narzędzia

Elementami modułu są między innymi klasy opisujące prymitywy geometryczne *SPPlane* oraz *SPCube*. Są to klasy rozszerzające *SPSceneElement*, które mogą zostać dodane do sceny w dwojaki sposób. Oba sposoby mogą są poprawne, jednakże pierwszy daje możliwość rozszerzania funkcjonalności klasy opisującej sześcian:

```
// (1)
class mojaKlasa : public SPCube(2.0f) { /*własne definicje*/ }

// (2)
class innaKlasa : public SPSceneElement
{
	public: innaKlasa(float dl_boku)
	{
		this->DisplayModel = SPCube::GetModel(dl_boku);
	}
};
```

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Kolejnym wartym wymienienia elementem modułu jest klasa *SPExtendedLinearMoving* udostępniająca prosty interfejs do generowania liniowych przekształceń jak przesuwanie, skalowanie czy obroty. Poniższy przykład ilustruje tworzenie przekształceń położenia w czasie:

```
SPExtendedLinearMoving X_POS;
bool wyczyscPoprzedniaSciezke = true;
X_POS.AddStep(SPLinearMoving(0.0f, 100.0f, 100.0f), !wyczyscPoprzedniaSciezke);
X_POS.AddStep(SPLinearMoving(100.0f, 200.0f, 200.0f),
                                                    !wyczyscPoprzedniaSciezke);

// pojazd przez pierwsze 100 ms przebędzie 100 jednostek długości
// oraz nastepne 100 przez kolejne 200ms
// funkcja GetValueFromDeltaTime zwraca odpowiednią wartość zależności od
// przyrostu czasu
void Pojazd::OnEnterFrame(SPRenderEvent * e)
{
	float x = X_POS.GetValueFromDeltaTime(e->frameTime);
	this->MoveTo( SPPoint3D(x,0.0f, 0.0f));
}
```

Choć nazwa klasy podpowiada wykorzystanie jej w animacjach przesunięć, to może ona zostać użyta również, przy operacjach obrotów obiektów czy też skalowania. Pozostałe klasy opisują obiekty takie jak wierzchołki czy wektory implementując operacje mnożenia, dodawania oraz liczenia długości, wyznaczania kątów czy wektora prostopadłego.

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Narzędzia dodatkowe

Silnik współczesnej gry komputerowej oprócz podstawowych modułów powinien zawierać pokaźną listę funkcjonalności (ang. features), aby był użytecznym narzędziem. Do najpopularniejszych funkcjonalności należą z pewnością mapy wysokości, mapy oświetlenia, dynamiczne cienie, obsługa *shaderów* czy fizyki. Na potrzeby *SimpleProject* wybrałem implementacje mapy wysokości z powodu jej uniwersalności, i stałej obecności w różnego typu grach trójwymiarowych. Poniższy rozdział został całkowicie poświęcony mechanizmowi generowania terenu z map wysokości, zawiera on opis problemu, implementację oraz przedstawienie wyników osiągniętych w ramach pracy inżynierskiej. Dużą część rozdziału poświęciłem omówieniu zagadnień związanych z wydajnością.

Mapy wysokości terenu

Jednym z ważnych elementów gry jest teren po którym porusza się gracz. Właściwie we wszystkich rodzajach gry istnieje wymaganie tworzenia mniej lub bardziej dokładnych odwzorowań terenu. Przypuśćmy, że zadaniem programistów jest stworzenie gry z gatunku FPS, używając potocznego języka *strzelanki*. Bohater gry jest żołnierzem przemierzającym rozległe i bardzo zróżnicowane tereny, od podmokłych nizin do surowych górzystych terenów. Przed twórcami stoi zadanie wyboru technologii modelowania ogromnych przestrzeni terenu, spełniającej poniższe założenia:

- Prosty sposób modelowania olbrzymich terenów
- Możliwość łatwego wprowadzania różnorodności ukształtowania
- Możliwość ustalenia dokładności odwzorowania
- Łatwe nakładanie tekstur
- Interfejs do pobierania wysokości w danym punkcie

Teoria

Mapa wysokości stosowania powszechnie w grafice komputerowej jest implementacją znanego od dawna przedstawiania terenu jako kolorowej mapy z dołączoną legendą mapującą kolor obszaru na wysokość n.p.m. Fotografia terenu czy też każdy inny obraz może być rozpatrywany jako element trójwymiarowy, jeśli przyjmiemy, że położenie danego piksela odpowiada położeniu na płaszczyźnie natomiast jasność koloru, wysokości danego punktu. Przykładem mapy wysokości planety Ziemia jest poniższy obraz:

Rys. 8 Mapa wysokości świata [30].

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Skala szarości powyższego obrazu wskazuje na wysokość terenu. Im ciemniejsze elementy obrazu tym wysokość jest mniejsza. Jaśniejsze elementy obrazu to Himalaje oraz Andy w Ameryce Południowej, co zapewne nie jest zjawiskiem zaskakującym. Dosyć jasna jest również Antarktyda, gdyż średnia wysokość skorupy lodu wynosi około 2040 m.

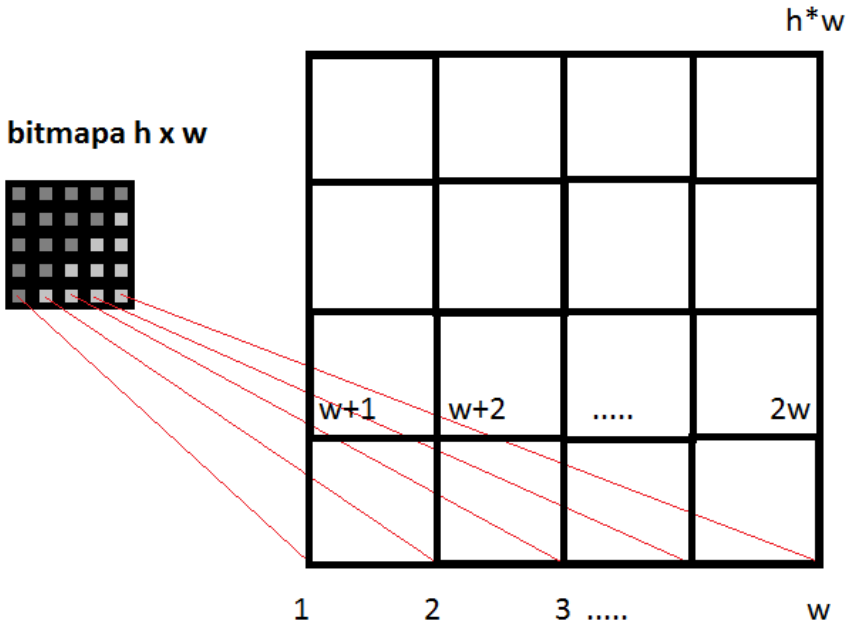
Warto zauważyć, iż kolor jako wartość skwantowana przybiera określoną liczbę wartości, w przeciwieństwie do wysokości która jest wartością ciągłą. Fakt ten nakłada ograniczenie na maksymalną dokładność takiej mapy jeśli chodzi o wysokość, ale również szerokość i długość. W obu przypadkach twórca gry powinien mieć możliwość ustalenia wielkości mapy, przed mapowaniem jej na konkretne wierzchołki sceny trójwymiarowej. Poniższy przykład ukazuje sposób tworzenia wierzchołka terenu:

```
// wysokość
float y_OpenGL = mapaWysokosci.PobierzJasnosc(x,z) * skalowanie_wysokosci;
// płaszczyzna
float x_OpenGL = x * skalowanie_płaszczyzny;
float z_OpenGL = z * skalowanie_płaszczyzny;
DodajWierzchołek ( Point3D(x_OpenGL, y_OpenGL , z_OpenGL ) );
```

W procesie tworzenia mapy terenu każdy pojedynczy piksel obrazu zamieniany jest na wierzchołek. Przestrzeń między wierzchołkami na odpowiednie płaszczyzny złożone z kwadratu lub dwóch trójkątów. Odległość między kolejnymi wierzchołkami na płaszczyźnie określa wartość skalowania. Im większa wartość skalowania tym obszar będzie większy, jednakże mniej dokładny ponieważ będzie zbudowany z coraz większych trójkątów. Alternatywnie powiększać teren bez straty dokładności można przez zwiększanie mapy wysokości co z kolei prowadzi do kwadratowego przyrostu wierzchołów. Należy zauważyć, iż już w przypadku bitmapy 100 na 100 pikseli generowanych będzie co najmniej 10 000 wierzchołów.

Zaimplementowane rozwiązanie działa dwuetapowo. W pierwszym kroku przeglądane są wszystkie piksele mapy i tworzone są wierzchołki zgodnie z powyższym przykładem. Drugim etapem jest przeglądanie wierzchołków w celu utworzenia płaszczyzn i wektorów normalnych. Poniższy rysunek przedstawia sposób reprezentacji wierzchołków związany z przeglądaniem od lewej wszystkich pikseli obrazu:

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29



Rys. 9 Mapowanie pikseli obrazu na wierzchołki terenu

Podczas tworzenia płaszczyzny iterator mapuje liniową tablicę zapisaną na dysku (bitmapa) na dwuwymiarową płaszczyznę:

```
// x i z to współrzędne kolejnych płaszczyzn
  int i0 = (x +  z   * Szerokosc_mapy)+1; // lewy-dolny (pierwszy iterator to 1)
  int i1 = (x +  z   * Szerokosc_mapy)+2; // prawy-dolny
  int i2 = (x + (z+1)* Szerokosc_mapy)+2; // prawy-górny
  int i3 = (x + (z+1)* Szerokosc_mapy)+1; // lewy-górny
```

Siatki trójkątów

Krokiem zwiększającym wydajność przetwarzania terenu jest zastosowanie siatek trójkątów. Jest to opcja pozwalająca niemalże osiągnąć wygenerowanie jednego trójkąta poprzez przesłanie do karty grafiki zaledwie jednego wierzchołka. Algorytm generowania siatki tworzy kolejne trójkąty z trzech ostatnio podanych wierzchołków. Aby uruchomić ten tryb renderowania można między innymi wywołać funkcję

```
glBegin(GL_TRIANGLE_STRIP)
```

Generowanie siatki może odbywać się zgodnie lub przeciwnie do ruchu wskazówek zegara, w zależności od argumentu odpowiedniej funkcji OpenGL.
W przypadku generowania map wysokości, poprzez przekazywanie do karty kolejnych trójkątów, obliczenia stają się czasochłonne z powodu wspomnianego już kwadratowego przyrostu ilości wierzchołków wraz z powiększaniem się mapy wysokości. Rozważany przykład dla kwadratowej mapy wysokości o boku sto pikseli wymaga przeliczenia 58 806 wierzchołków:

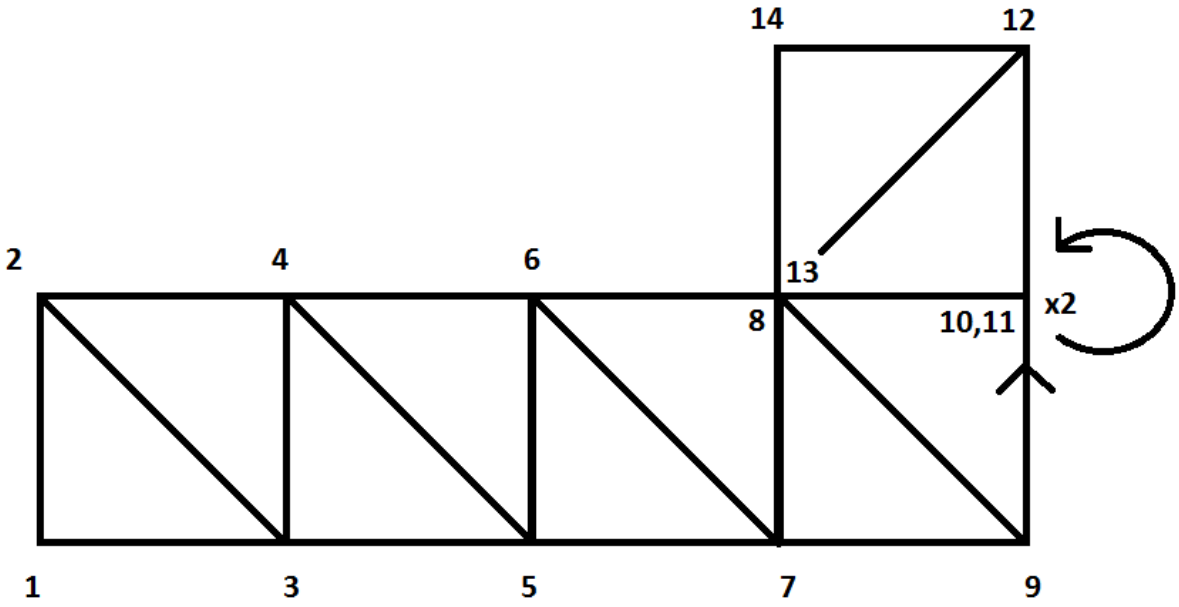
$$liczbaWierzchołków = (100 - 1)^2 * 2 * 3$$

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Natomiast przy zastosowaniu siatek trójkątów zaledwie 19 602, w ogólnym przypadku trójkąt powstaje poprzez dodanie jednego wierzchołka.

$$liczbaWierzchołków = (100 - 1)^2 * 2$$

Podczas tworzenia siatki trójkątów należy zwrócić uwagę na kolejność generowania kolejnych wierzchołków. Ma to istotny wpływ na określenie przez kartę grafiki przodu i tyłu trójkąta. Aby uniknąć problemu z kolejnymi wierszami siatki można zrezygnować z tworzenia spójnej jednolitej sieci, i generować wiele pasków trójkątów. Alternatywnie podczas przechodzenia do kolejnego wiersza można wygenerować dodatkowy trójkąt w punkcie poprzez dodanie dodatkowego wierzchołka. Takie rozwiązanie zostało przedstawione na poniższym rysunku, w momencie przejścia do kolejnego wiersza tworzony jest wierzchołek o numerze 11 dokładnie w tym samym miejscu co wierzchołek 10.



Rys. 10 Sposób generowania siatki trójkątów

Powyższy algorytm w celu wygenerowania terenu z mapy wysokości 100 x 100 przesyła do karty graficznej 19 800 wierzchołków:

$$liczbaWierzchołków = (100 - 1) * (1 + 1 + (100 - 1) * 2)$$

Ukrywanie niewidocznych płaszczyzn

Generowanie terenu jest jednym z najbardziej obciążających sprzęt zadań z racji liczby wierzchołków rzędu dziesiątek tysięcy. Aby ograniczyć w sposób znaczący obliczenia, można stosować kilka metod:

- odrzucanie płaszczyzn odwróconych tyłem
- odrzucenie elementów zasłanianych przez inne obiekty
- odrzucanie niewidocznych fragmentów (ang. view culling frustrum)

Powyższe metody różnią się stopniem komplikacji oraz zyskiem jaki mogą przynieść. Ważnym elementem dalszych rozważań jest oszacowanie, czy w stosunku do stopnia trudności implementacji otrzymany efekt będzie zadowalający. Przykładem może być

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

pierwsza metoda - odrzucanie płaszczyzn odwróconych tyłem. Jest to skomplikowana metoda obarczana wieloma niedogodnościami. Pierwszą z nich jest fakt przetrzymywania zbiorów wierzchołków jako tablice zmiennych. Tablica jest generowana podczas tworzenia mapy wysokości i renderowana. Chęć pozbycia się niektórych wierzchołków musiałby wiązać się z przebudową tablicy w każdej klatce animacji, co obarcza procesor operacjami alokacji i zwalniania pamięci. Odrzucanie płaszczyzn odwróconych tyłem mogłoby zostać łatwo zaimplementowane przy użyciu funkcji *glVertex3f* poprzez przekazywanie kolejnych wierzchołków, jednakże mechanizm jest o rząd do dwóch mniej wydajny niż tablice wierzchołków. Tak niepewny zysk związany z powiększeniem złożoności obliczeniowej dla procesora, kosztem obliczeń karty graficznej nie odznaczyłby się jednak zauważalnym wzrostem wydajności. Spowodowane jest to charakterystyką terenu. Znakomita większość płaszczyzn jest bowiem skierowana ku górze a więc przodem do kamery, jedynie tylna część wzniesień zostałaby odrzucana. Dla terenu równinnego algorytm prawdopodobnie pogorszyłby wydajność. Dodatkowo pozostaje problem badania obiektów wklęsłych które pomimo pozostawania byłem do kamery powinny być wyświetlane. Drugim wymienionym sposobem jest analiza przesłonięć dostępna w bibliotece OpenGL od wersji 1.5. Biblioteka daje możliwość nałożenie na obiekty prymitywów obejmujących cały obiekt w celu uproszczenia obliczeń a następnie przeprowadzenie testu wyświetlania [42]. Biblioteka pozwala na wykonanie zapytania w celu rozpoczęcia analizy przesłonięć. Przy pomocy funkcji *glGetQueryObjectiv* można odczytać wynik zapytania, a następnie zdecydować czy dany obiekt będzie wyświetlany. Poniższy kod [42] przedstawia sposób wywołania zapytania i odczytania wyniku:

```
// analiza przesłaniania skomplikowanego obiektu
glBeginQuery( GL_SAMPLES_PASSED, query_id[ 0 ] );
// sześcian otaczający obiekt
glCallList( CUBE_0 );
glEndQuery( GL_SAMPLES_PASSED );

// skierowanie poleceń do wykonania
glFlush();
//oczekiwanie na wynik
GLint available;
do
{
    glGetQueryObjectiv( query_id[ 0 ], GL_QUERY_RESULT_AVAILABLE, & available);
}while( !available );

GLint result;
glGetQueryObjectiv( query_id[ 0 ], GL_QUERY_RESULT, & result );
if( result )
// rysowanie obiektu
       glCallList( SAMOCHOD_0 );
```

Trzecim rozważanym elementem jest zastosowanie wbudowanej w silnik *SimpleProject* funkcjonalności odrzucania fragmentów poza bryłą widzenia. Integracja okazała się bardzo prosta, ponieważ wystarczającą modyfikacją jest podzielenie jednego obiektu jakim jest cały teren, na mniejsze części, i dodanie ich do drzewa ósemkowego obiektu *SPRenderer*. Efekty działania mechanizmu z pewnością wpłyną pozytywnie na wydajność, ograniczając listę wyświetlanych elementów, w szczególnych przypadkach jak skierowanie kamery w górę, do

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

zera. Średni wynik algorytmu jest również zadowalający gdyż odrzuca wszystkie elementy z tyłu kamery, oraz częściowo po bokach, a więc skuteczność przewyższa 50%.

Teksturowanie

Teksturowanie całości modelu utworzonego z mapy wysokości, polegające na określaniu współrzędnych teksturowania każdego wierzchołka, było by czasochłonne i trudne, lecz dające grafikom pełną władzę nad wynikami pracy. Alternatywnym pomysłem jest nałożenie całej tekstury na powierzchnie. Taka operacja wprowadza pewną dozę przypadkowości lecz w ogólnej ocenie jest akceptowalna. Defekty uwidaczniają się przy wielkich wypiętrzeniach terenu gdzie dany fragment tekstury jest rozciągany do stosunkowo dużych rozmiarów tracąc na jakości. Sposób określania współrzędnych tekstur jest stosunkowo prosty i możliwy do zapisania w zaledwie jednej linii kodu:

```
// wys i szer to wysokość i szerokość mapy w pikselach
for(int z = 0 ; z <= wys; z++)
	for(int x = 0 ; x <= szer; x++)
		WspolrzednaTekstury.push_back( Point2D( x / szer, z / wys ) );
```

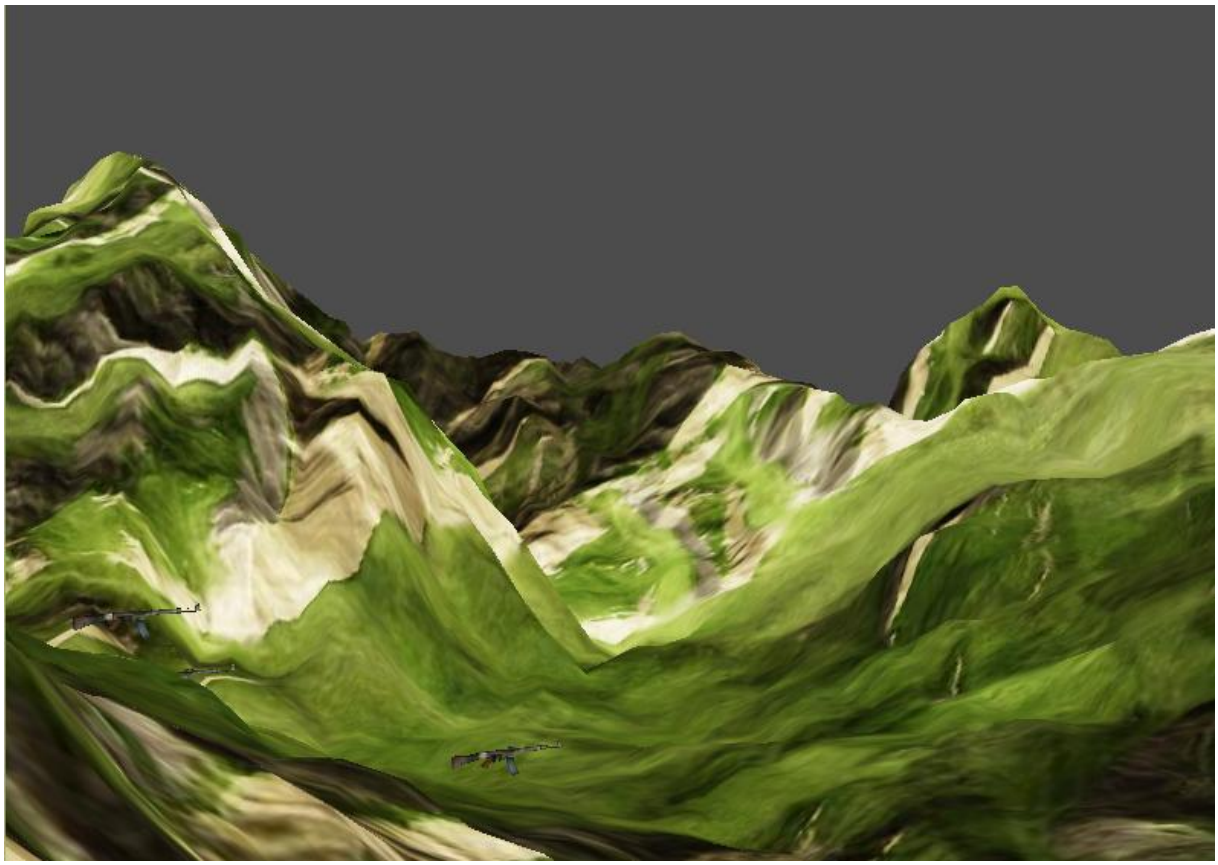
W bibliotece OpenGL współrzędne teksturowania dwuwymiarowego określa się przez podanie dla każdego wierzchołka odpowiadającej mu współrzędnej tekstury z zakresu [0 .. 1]. W celu podziału tekstury na odpowiednią ilość wystarczy podzielić kolejne współrzędne przez ich maksymalne wartości.

Implementacja i wyniki

Zaimplementowane narzędzie do tworzenia terenu udostępnia użytkownikowi prosty interfejs złożony z konstruktora i dwóch funkcji do pobierania wartości wysokości:

```
//tworzenie mapy na podstawie obiektu SPTexture hmap
//nałożona tekstura jest przekazywana w drugim parametrze
//teren składa się z części zawierajacych 50x50 wierzchołków o
//odstępach w każdym kierunku równym jednej jednostce
HMap *h = new HMap(hmap, hmapTex, 50, SPPoint3D(1.0f, 1.0f, 1.0f));
//przesunięcie terenu aby najniższy punkt miał wysokość 0
h->Move(SPPoint3D(0.0f,-(h->GetMinPos()),0.0f));

//pobieranie wysokości
float HMap::GetHeight(float x, float z);
void HMap::GetHeight(Point3D *p); // wysokość wpisywana w pole p->y
```

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29



Rys. 11 Mapa wysokości z nałożoną teksturą

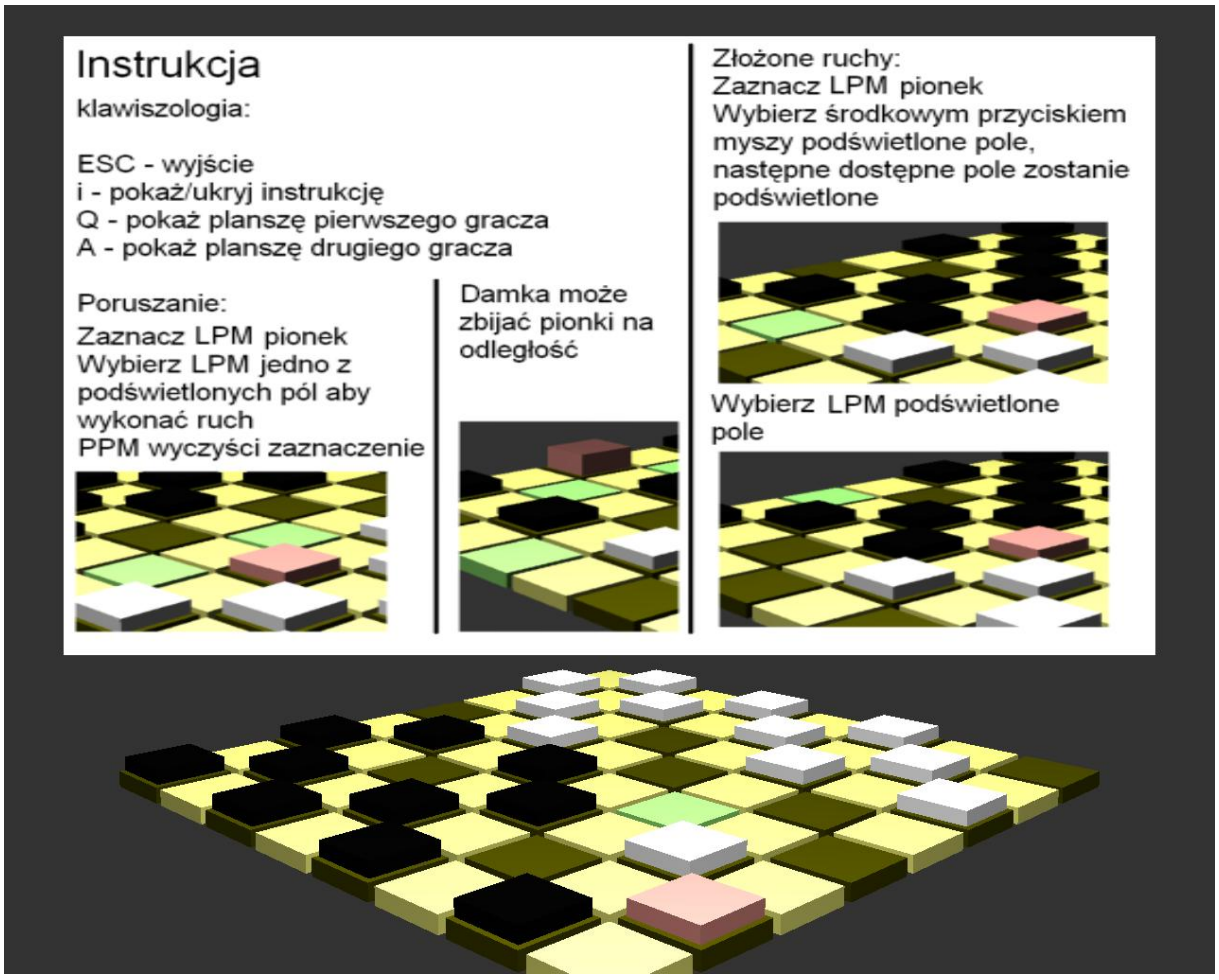
Rys. 12 Rozciąganie tekstury na stromych zboczach

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Przykład gry

Gra warcaby

Gra w klasyczne warcaby jest przeznaczona dla dwóch graczy których celem jest zbicie pionków przeciwnika. Sterowanie odbywa się przy użyciu myszy i klawiatury. Jest to pierwsza gra która powstała w celu weryfikacji przydatności i niezawodności elementów silnika.


Instrukcja
klawiszologia:
ESC - wyjście
i - pokaż/ukryj instrukcję
Q - pokaż planszę pierwszego gracza
A - pokaż planszę drugiego gracza
Poruszanie:
Zaznacz LPM pionek
Wybierz LPM jedno z podświetlonych pól aby wykonać ruch
PPM wyczyści zaznaczenie
Damka może zbijać pionki na odległość
Złożone ruchy:
Zaznacz LPM pionek
Wybierz środkowym przyciskiem myszy podświetlone pole, następne dostępne pole zostanie podświetlone
Wybierz LPM podświetlone pole

Rys. 13 Gra warcaby. Instrukcja

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Tworzenie obiektów gry

Interfejs silnika umożliwia tworzenie wlanych klas opisujących obiekty sceny, oraz integracje ich z silnikiem gry, umożliwiając przetwarzanie zdarzeń, wyświetlanie w scenie czy komunikacje z innymi obiektami. Rozszerzanie funkcjonalności klas odbywa się przez dziedziczenie z klasy SPSceneElement [1]:

```
#include "SimpleProject.h"
class Pionek : public SPSceneElement
{
	int x;
	int y;
	void OnPress(SPMouseEvent *e);
	void OnKeyUp(SPKeyEvent *e);
	void OnEnterFrame(SPRenderEvent * e);
};
```

Wbudowane prymitywy geometryczne

Wyświetlane obiekty są przechowywane w module renderowania, który jest udostępniany przez funkcje klasy *SPRenderer*. W prosty sposób obiekty dziedziczące ze wspomnianej wyżej klasy mogą zostać dodane do listy wyświetlania w jednej z trzech konfiguracji: wyświetlane zawsze, przechowywane w drzewie ósemkowym lub wyświetlane we współrzędnych kamery. Silnik implementuje podstawowe prymitywy geometryczne [2]:

```
Pionek *pionek = new Pionek();
pionek->DisplayModel = SPCube::GetModel(0.8f);
SPRenderer * renderer = SPRenderer::GetInstance();
renderer->AddToRendering(pionek, SPRenderer::ALWAYS);
```

Obsługa zdarzeń

Obsługa zdarzeń generowanych przez kontrolery gry jest możliwa dzięki zarejestrowaniu obiektu jako nasłuchiwacza zdarzeń. Silnik obsługuje zdarzenia wciskania i zwalniania klawiszy myszy oraz klawiatury, klikania myszą na obiekt, zwalniania klawisza myszy nad obiektem, zdarzenie wyświetlania klatki animacji itd. [3].

```
void Pionek::OnPress(SPMouseEvent *e)
{
	if(this->Aktywny && e->Button == SPMouse::LEFT)
	{
		this->plansza->OdznaczWszystkiePolaOrazPionki();
		this->plansza->ObliczMozliwosciRuchu(this);
		this->Zaznacz();
	}
}
SPApplication *app = SPApplication::GetInstance();
app->Events.OnPress.AddHandler(&SPSceneElement::OnPress, pionek );
```

[1] zob. str. 32
[2] zob. str. 40
[3] zob. str. 32 oraz 22

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Animowanie ruchu metodą interpolacji liniowej

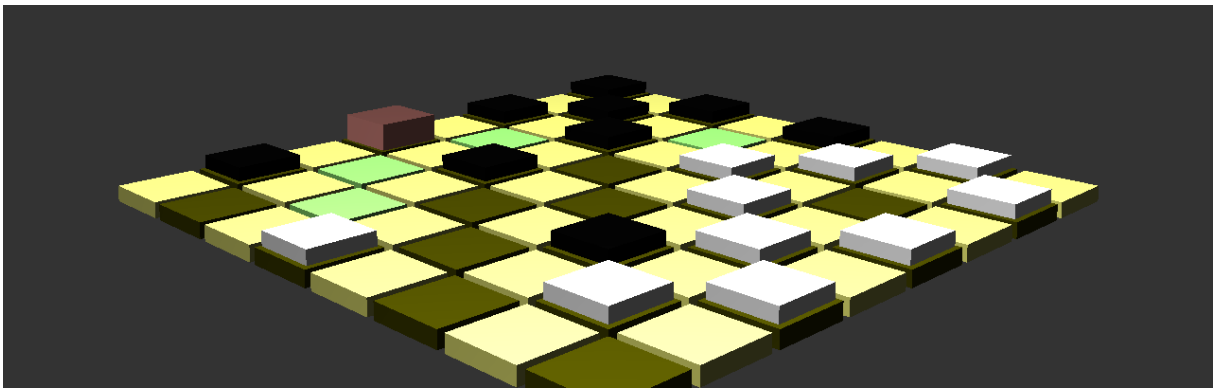
W grze zaimplementowano poruszanie się pionków metodą interpolacji liniowej. Ścieżka ruchu może składać się z kilku elementów jeśli przy odpowiednim ustawieniu pionków przeciwnika gracz ma możliwość kilkukrotnego bicia. Za generowanie położenia pionków na mapie odpowiada klasa *SPExtendedLinearMoving* będąca elementem silnika [4].

```
class Pionek : public SPSceneElement
{
	SPExtendedLinearMoving X_POS;
	SPExtendedLinearMoving Z_POS;

void Pionek::OnEnterFrame(SPRenderEvent * e)
{
	SPPoint3D pos(X_POS.GetValueFromDeltaTime(e->frameTime),0.0f
				,Z_POS.GetValueFromDeltaTime(e->frameTime));
	this->MoveTo(pos);
}
//                          x_ początek ,      x_koniec ,          czas [ms]
X_POS.AddStep(SPLinearMoving(     0.0f,         10.0f,               100.0f));
X_POS.AddStep(SPLinearMoving(     10.0f,        50.0f,                 20.0f));
X_POS.AddStep(SPLinearMoving(     50.0f,        20.0f,               200.0f));
```

Skalowanie, przesuwanie i obracanie obiektów

Podczas rozgrywki prócz przesuwania obiektów sceny, wykonywane są operacje skalowania do oznaczenia *damki* oraz obrotów do animacji zbijanych pionków jak i ruchów kamery. Podstawowe funkcjonalności dostarcza klasa *SPSceneElement* z której muszą dziedziczyć elementy gry [5].

```
pionek->MoveTo      (SPPoint3D p);
pionek->Move        (SPPoint3D p);
pionek->SetScale    (SPPoint3D p);
pionek->SetRot      (SPPoint3D p);
pionek->Rotate      (SPPoint3D p);
```



Rys. 14 Gra warcaby. Skalowanie *damki*

[4] zob. str. 32
[5] zob. str. 32

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Scena. Pobieranie obiektów sceny

Niezbędnym elementem każdej gry jest komunikacja pomiędzy obiektami sceny. Funkcjonalność może zostać zaimplementowana poprzez przekazywanie wskaźników na obiekty lecz rozwiązanie to wydaje się niewygodne przy większej ilości elementów sceny. Silnik udostępnia abstrakcję sceny rozumianej jako kontener obiektów. Elementy należące do danej instancji klasy *SPScene* mogą komunikować się wzajemnie [6]:

```
SPScene  *Scena1 = new SPScene();
Scena1->AddElementToScene("pionek1",pionek);
Scena1->AddElementToScene("plansza1",plansza);

void Pionek::OnPress(SPMouseEvent *e)
{
	Plansza * p = dynamic_cast<Plansza*>(
			this->GetScene()->GetElementById("plansza1"));
	if(p != NULL)
		p->ZaznaczonyPionek = this;
}
```

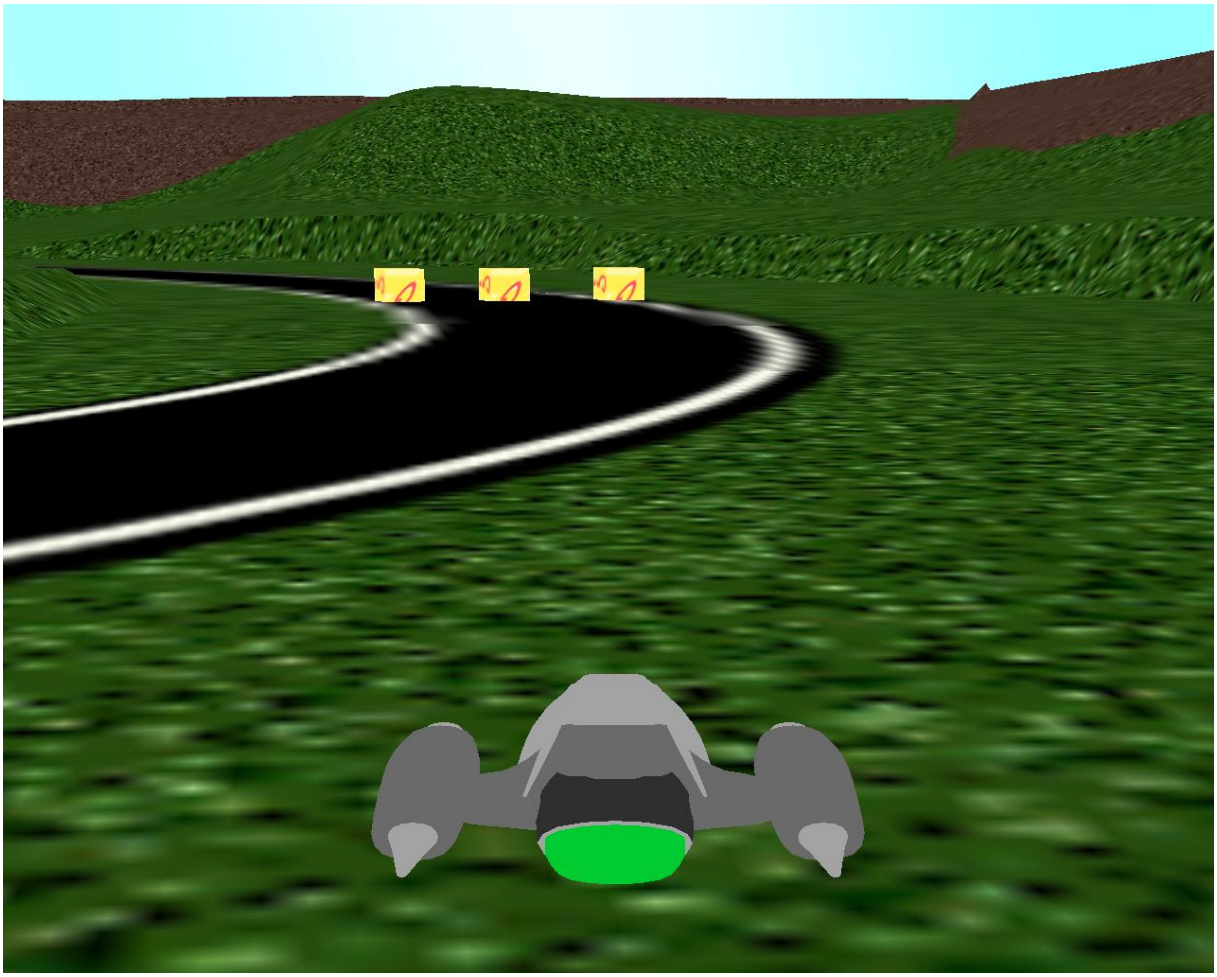
[6] zob. str. 32

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Gra wyścig

Gra wyścig jest przeznaczona dla jednego gracza. Celem gry jest pokonanie trasy w jak najkrótszym czasie. Po przejechaniu odpowiedniej ilości okrążeń na ekranie zostaje wyświetlony czas przejazdu. Dodatkowo na mapie rozmieszczono 9 paczek, w których losowo umieszczone zostały:

- Przyspieszenie – pojazd przez 4 sekundy może dwukrotnie przekraczać maksymalną prędkość
- Spowolnienie – gracz przez 4 sekundy nie może przekroczyć połowy maksymalnej prędkości
- Tarcza – chroni gracza przed jedną paczką ze spowolnieniem



Rys. 15 Gra wyścig. Kadr z gry

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Proxy tekstur i modeli 3D

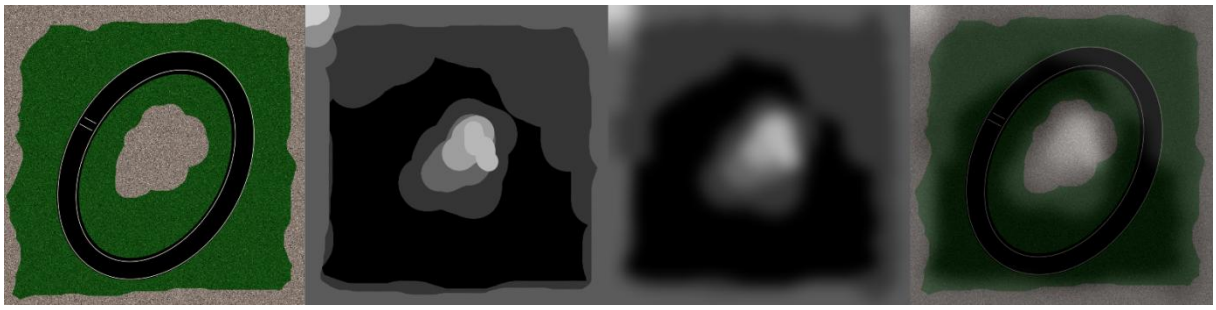
W grze został wykorzystany model pojazdu [43]. Silnik udostępnia prosty interfejs do wczytywania plików .obj, .mtl oraz .bmp. Dostęp do zasobów udostępnia model *Pośrednika* (ang. proxy):

```
SPTextureProxy * proxy = SPTextureProxy::GetInstance();
SPTexture *paczka_tex;
bool synchronicznie = false;
ResourceItem< SPTexture*,SPSceneElement*> res =
proxy->LoadResource(
       „paczka.bmp”,        // nazwa pliku z zasobem
       NULL,                // wskaźnik na obiekt (tylko dla funkcji asynch.)
       synchronicznie       // funkcja blokująca
);
paczka_tex = res.Resource;
```

Funkcja wczytywania występuje jako funkcja blokująca wczytująca zasób synchronicznie lub umożliwia wczytywanie asynchroniczne. Przy wyborze wczytywania asynchronicznego należy podać wskaźnik na obiekt, do którego zasób będzie przekazany jako argument zdarzenia. Teksturowanie obiektów sceny odbywa się automatycznie zgodnie z współrzędnymi teksturowania odczytanymi z modelu [7].

Mapa wysokości

Na potrzeby gry wyścig powstała tekstura powierzchni oraz mapa wysokości przy użyciu programu *Paint.Net*. Tekstura trawy i skały powstała przy użyciu efektu szumu oraz wyborze odpowiednich kolorów. Następnie na podstawie stworzonej miniatury (rys. 16.1) powstała wstępna mapa wysokości (rys. 16.2), na który został nałożony efekt rozmycia metodą Gaussa (rys. 16.3). Obrazek czwarty przedstawia złożenie mapy wysokości oraz miniatury tekstury. Tekstura została przygotowana w rozdzielczości 2048x2048px, natomiast mapa wysokości 200x200px.



Rys. 16 Gra wyścig. Proces powstawania terenu

Silnik udostępnia prosty interfejs tworzenia mapy wysokości:

[7] zob. str. 25

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

```
HMap *teren = new HMap(
       hmap,         // mapa wysokości – wskaźnik na SPTexture
       hmapTex,      // tekstura – wskaźnik na SPTexture
       50,           // podział mapy na 16 elementów po 50x50 wierzchołków
       SPPoint3D(1.0f, 1.0f , 1.0f )     // skalowanie x ,y, z
);
```

Parametr podziału mapy jest używany do optymalizacji wyświetlania, mapa wysokości 200x200px (40 000 wierzchołów) jest dzielona na 16 obiektów 50x50px (po 2500 wierzchołków). Dzięki mechanizmom klasy *SPRenderer* wyświetlane są tylko te widoczne na ekranie. Tekstura jak i mapa wysokości powinna być kwadratem o długości boku będącą potęgą liczby 4 [8].

Sky Box (Sphere)

Niezbędnym elementem sceny jest skrzynka lub sfera imitująca niebo. Przypisanie wskaźnika na obiekt *SPTexture* do jednej ze zmiennych klasy *SPRenderer* spowoduje wyświetlanie oteksturowanej sfery wokoło kamery. Wielkość sfery jest ustawiana proporcjonalnie do wielkości tekstury, natomiast samo wyświetlanie obywa się z wyłączonym buforem Z, aby tekstura sfery była renderowana nieskończenie daleko [9].

```
renderer->skyBoxTex = niebo;      // przypisanie wskaźnika na obiekt SPTexture
```

Pliki konfiguracyjne

Podczas testowania i poprawiania detali gry, kompilowanie po każdej zmianie byłoby uciążliwym zadaniem. Silnik dostarcza interfejs do odczytywania prostych plików konfiguracyjnych. W grze wyścigi możliwość konfigurowania obejmuje, położenia obiektów, szybkości poruszania pojazdu, czy ilości okrążeń toru do zakończenia wyścigu:

```
//conf2.txt
#komentarz poprzedzamy znakiem „#”
model_name spaceship.obj
czas_trwania_bonusu_ms 4000.0f
start_pos -69.3f;0.2f;-51.5f
laps 1

//sGra.cpp
SPConfig *conf_file = SPConfig::GetInstance();
conf_file->ReadFile("conf2.txt");
std::vector<float> start_pos_samochod =
                    conf_file->ReadAsFloatVector("start_pos");
float czas_trwania_bonusu_ms = conf_file->ReadAsFloat("czas_trwania_bonusu_ms");
std::string name_model = conf_file->Read("model_name");
int laps = conf_file->ReadAsInt("laps");
```

[8] zob. str. 42
[9] zob. str. 31

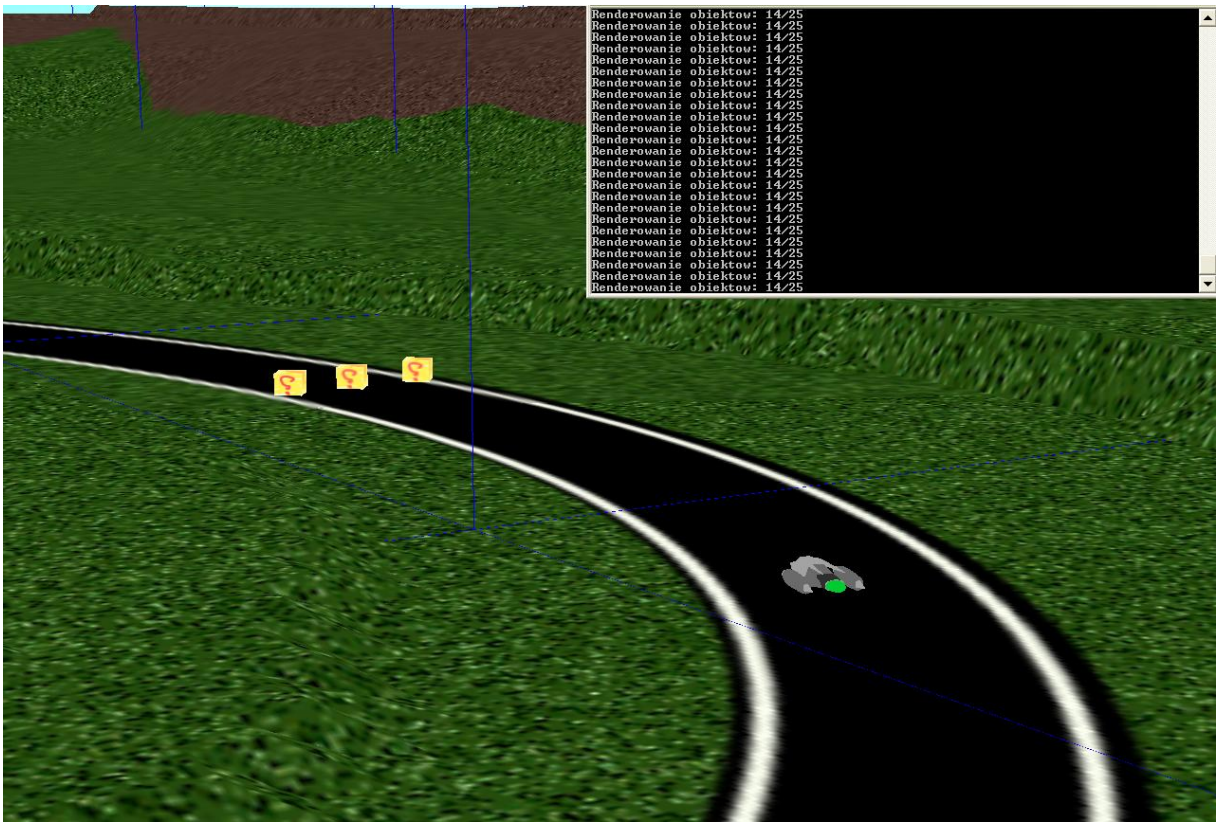
Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Rendering. Drzewo obiektów i Frustrum Culling

W grze wyścig występuje stosunkowo duża ilość wierzchołków do wyświetlania:

- Pojazd – poniżej 1000 (wyświetlany zawsze)
- Wyświetlanie czasu - 28 (siedem obiektów, przez większość gry ukrytych)
- Paczki – 72 = (dziewięć paczek wyświetlanych tylko w bryle widzenia)
- Teren – 40 000 (szesnaście obiektów wyświetlanych tylko w bryle widzenia)

Poniższy obrazek ilustruje scenę wraz z drzewem ósemkowym. W scenie znajduje się 25 obiektów – 9 paczek oraz 16 elementów terenu (nie licząc pojazdu), obecnie karta grafiki przetwarza jedynie 14 obiektów (3 paczki oraz 11 elementów terenu) czyli 27 524 z 40 072 wierzchołków [10].



Rys. 17 Gra wyścig. Zastosowanie drzewa ósemkowego do ukrywania niewidocznych obiektów

Aby obiekty były przechowywane w drzewie i renderowane jedynie gdy znajdują się w bryle widzenia należy dodać je do renderowania z parametrem *TREE*

```
renderer->AddToRendering(paczka[i] , SPRenderer::TREE);
```

[10] zob. str. 28

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Gra snajper

Gra snajper jest przeznaczona dla jednego gracza. Celem jest zdobycie jak największej liczby punktów za strzelanie do skrzynek.

Rys. 18 Gra snajper. Ekran wczytywania. Jako tło użyto kadru z filmu *Szeregowiec Rayan*

Asynchroniczne wczytywanie zasobów

W grze strzelnica zastosowano funkcjonalność asynchronicznego wczytywania zasobów, ekran wczytywania wyświetla pasek postępu, podczas gdy w tle kolejne wątki odczytują zasoby zapisane na dysku. Fakt wczytania zasobu jest dostarczany do obiektu jako zdarzenie [11]:

[11] zob. str. 25

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

```
EkranWczytywania * ekranWczytywania  = new EkranWczytywania();
SPTextureProxy * proxy = SPTextureProxy::GetInstance();
SPModelProxy * modelProxy = SPModelProxy::GetInstance();
bool asynchronicznie = true;
proxy->LoadResource(
„drzewo.bmp”,              // nazwa pliku z zasobem
ekranWczytywania,          // wskaźnik na obiekt
asynchronicznie            // funkcja nieblokująca
);
modelProxy->LoadResource(„drzewo.obj”,  ekranWczytywania,  asynchronicznie );

void EkranWczytywania::OnResourceLoaded(void * ptr, std::string id)
{
       if(id == "drzewo.bmp")
       {
              this->drzewo_tex = (SPTexture*)ptr;
              this->postep++;
       }
       else if(id == "drzewo.obj")
       {
              this->drzewo_model = (SPModel *)ptr;
              this->postep++;
       }
       …
       …
}
```

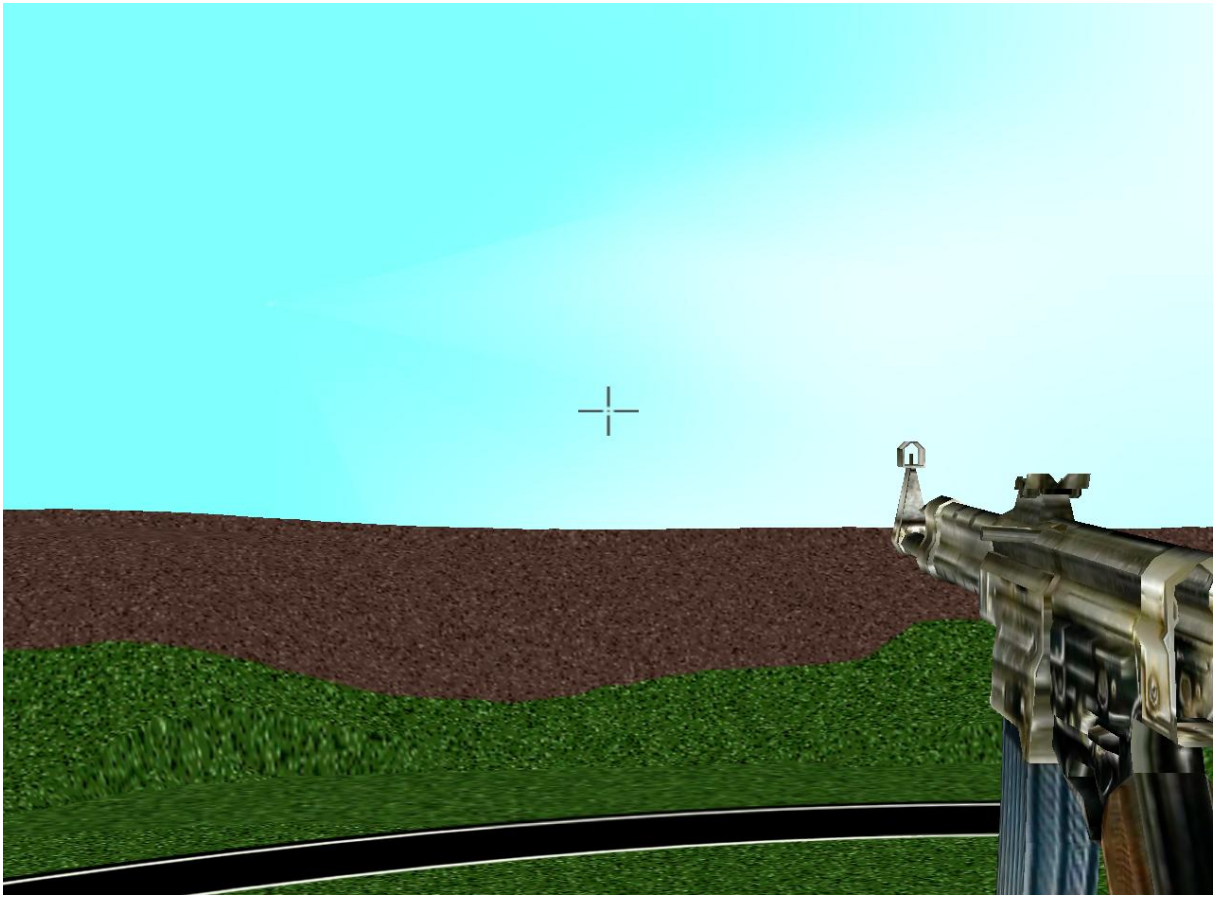
Rendering. OnGUI

Gra snajper wykorzystuje interfejs silnika do renderowania obiektów we współrzędnych kamery. Położenie obiektu wyświetlanego w trybie *OnGui* jest uzyskane przez odłożenie na stos macierzy projekcji, macierzy jednostkowej. Poniższy przykład ilustruje wyświetlenie obiektu *celownik* w centrum ekranu oraz *karabin* w prawym dolnym rogu [12].

```
renderer->AddToRendering(celewnik , SPRenderer:: ONGUI);
renderer->AddToRendering(karabin , SPRenderer:: ONGUI);
celownik->MoveTo(SPPoint3D( 0.0f, 0.0f, - renderer->Camera->Near ));
karabin->MoveTo(SPPoint3D(renderer->Camera->Horizontal/2.0f,
              -renderer->Camera->Vartical/2.0f, - renderer->Camera->Near ));
```

[12] zob. str. 25

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29



Rys. 19 Gra snajper. Widok broni

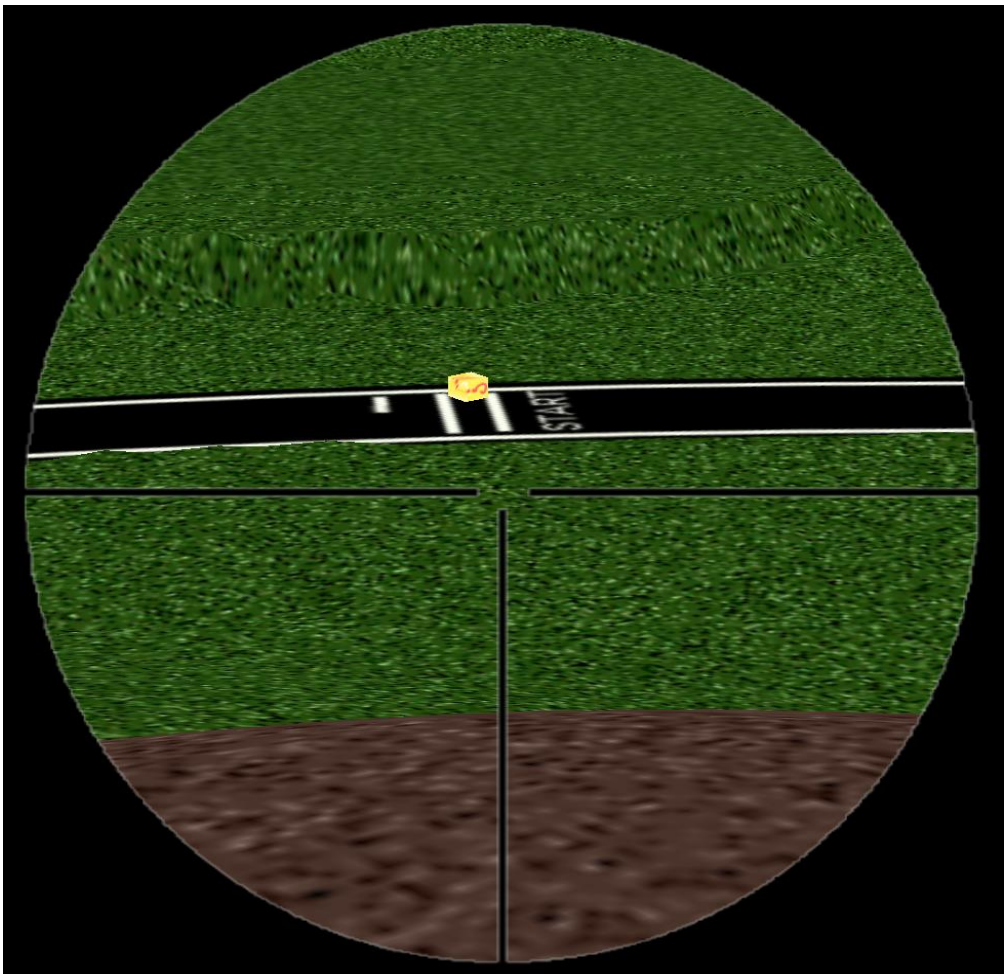
Rendering. Przeźroczystość, maska

Do wyświetlania celownika oraz widoku przez lunetę została wykorzystana metoda biblioteki *OpenGL* mieszająca piksele przechowywanych w dwóch buforach ramki: *glEnable(GL_BLEND)*. Funkcjonalność biblioteki pozwala na określenie funkcji mieszającej. Obiekty sceny udostępniają prosty interfejs włączenia mieszania poprzez ustawienie flagi [13]:

```
SPSceneElement * celownik = new SPSceneElement();
// piksele obiektu celownik będą mieszane z pikselami tła
celewnik->blend = true;
```

[13] zob. str. 28

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29



Rys. 20 Gra snajper. Widok przez lunetę

Kamera – ffp

Silnik gry udostępnia gotową kamerę sterowaną myszą oraz strzałkami. Zależnie od położenia myszy liczone są kąty obrotu kamery i wektor po jakim powinna się poruszać kamera. Kamera udostępnia gotowe algorytmy do obliczania obrotów na podstawie pozycji myszy oraz kierunku poruszania na podstawie obrotów kamery:

```
SPPoint3D FPP_Camera::GetRotationFromMousePos(SPPoint2D p);
SPVector3D FPP_Camera::GetVectorFromRotations(SPPoint3D rot);
```

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Optymalizacje. Tablice wierzchołków

Wszelkie obiekty wyświetlane na scenie są przekazywane do karty graficznej w postaci nieindeksowanej, mieszanej tablicy wierzchołków. Silnik posiada funkcjonalność tworzenia tablic z modeli zapisanych w plikach .obj oraz konstruowanych przez użytkownika w kodzie gry [14].

```
SPModel *m = new SPModel();
SPDisplayObject obj ;
obj.Triangles.push_back(t1);
obj.Triangles.push_back(t2);
obj.Triangles.push_back(t3);
obj.CreateVertexTable();   // Tworzenie tablicy wierzchołków
m->Objects.push_back(obj);
m->UpdateBSphere();        // Generowanie sfery okalającej cały model
```

[14] zob. str. 10

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Zakończenie

W ramach tej pracy powstał prosty w obsłudze silnik gry, zrealizowany jako biblioteka języka C++. Jest ona narzędziem ułatwiającym tworzenie wachlarza rozwiązań, od gier strategicznych do zręcznościowych. Założeniem projektu było wydanie biblioteki nieukierunkowanej na konkretne rozwiązanie, lecz umożliwiającej rozwój, poprzez pisanie komponentów używających interfejsu silnika. Jednym z takich rozwiązań możliwych do zaimplementowania bez dostępu do kodu biblioteki jest napisany przez autora moduł generowania map wysokości. Ostatni rozdział zwięźle referuje elementy które udało się zawrzeć w tej implementacji, oraz przedstawia perspektywy dalszego rozwoju projektu. Rozdział wieńczy krytyczne spojrzenie na implementacje.

Podsumowanie prac

Jedną z pierwszych i najważniejszych funkcjonalności silnika jest umożliwienie obsługi zdarzeń kierowanych do aplikacji przez użytkownika oraz system operacyjny. Moduł zdarzeń biblioteki implementuje mechanizm przekazywania zdarzeń wprost do obiektów gry. Najczęściej stosowanymi kontrolerami gry są mysz i klawiatura, z tego powodu zaimplementowana została ich pełna obsługa . Szczególną uwagę należy poświęcić mechanizmowy obsługi zdarzenia wciśnięcia przycisku myszy na obiekcie. Został on zaimplementowany dzięki możliwością jakie daje biblioteka *OpenGL* poprzez renderowanie w trybie selekcji. Kolejną klasą zdarzeń jakie istnieją w silniku są zdarzenia emitowane przez obiekty. Programiści mają możliwość implementować komunikację między różnymi obiektami sceny dzięki mechanizmowi zdarzeń niestandardowych, przekazując im odpowiednie parametry, mogą sterować odpowiednio obiektami w grze. Zdarzenia niestandardowe jak i wszelkie inne mogą być rozsyłane do wszystkich obiektów lub do wyspecyfikowanego odbiorcy. Trzecią klasą zdarzeń są zdarzenia wewnętrzne silnika związane z cyklem życia obiektów – jak *OnStart*, *OnDelete*; czy ładowaniem zasobów modeli trójwymiarowych – *OnModelLoaded*. Wszelkie powyższe zdarzenia są kierowane do obiektów dziedziczących z klasy bazowej elementów sceny. Programiści mają możliwość łatwej obsługi zdarzeń poprzez implementacje wirtualnych funkcji klasy bazowej. Przekazywane argumenty są zgodne z typem zdarzenia, a biblioteka dba o wywoływanie funkcji obsługi z poprawnymi argumentami.

Zastosowanie hierarchicznej struktury obiektów w grze oraz zastosowanie zarządcy obiektów pozwala programistom silnika na bardziej intuicyjną pracę nad obiektami. Operacje obrotów, przesuwania oraz skalowania są propagowane z rodzica na dzieci dzięki czemu praca programisty została znacznie ułatwiona. Dodatkowo zarządca obiektów pozwala na pobieranie wskaźników na obiekty wewnątrz kodu innych obiektów, oraz wykonywanie ich funkcji zgodnie z zasadami enkapsulacji.

Odpowiedzią na zapotrzebowanie wykonywania długotrwałych operacji podczas użytkowania gry jest moduł zadań asynchronicznych. Przykładem wykorzystania modułu jest problem dynamicznego ładowania zasobów w trakcie gry. Zdarza się, iż ograniczona pojemność pamięci operacyjnej nie pozwala na przechowywanie wszystkich elementów gry, dlatego biblioteka udostępnia programiście interfejs do wykonywania długich operacji, jak na przykład ładowanie modeli trójwymiarowych z dysku. Mechanizm opiera się na wielowątkowym modelu Pośrednika (ang. proxy). Zarządca zasobów sprawdza czy dany zasób już znajduje się w pamięci, czy trzeba go pobrać. W obu przypadkach funkcja przypisania kończy się natychmiast nie blokując pętli gry. Zarządca przypisuje zasób w następnym obiegu pętli lub gdy pobierze go z dysku. Zastosowanie modelu *proxy* wprowadza również optymalizację, gdyż powołanie wielu obiektów korzystających z tego samego zasobu nie powoduje kopiowania zasobu wiele razy w pamięci operacyjnej. Kolejnym zadaniem

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

modułu zasobów graficznych jest specyfikacja interfejsu do obiektów bitmap i modeli trójwymiarowych stworzonych w programach graficznych. Na potrzeby tej pracy został zaimplementowany moduł tworzenia modeli, materiałów oraz teksturowania obiektów utworzonych z plików .obj oraz .mtl. Dodatkowo zaimplementowana została funkcjonalność pobierania obrazu z plików .bmp.

Ostatnim, wspomnianym już modułem jest przykład komponentu jaki może zostać napisany bez dostępu do kodu źródłowego silnika. Na potrzeby tej pracy został zaimplementowany moduł tworzący oteksturowany teren z pliku zawierającego mapę wysokości. Możliwość utworzenia tego modułu bez dostępu do kodu źródłowego silnika świadczy o elastyczności rozwiązania. Programista przy użyciu interfejsu silnika może konstruować własne moduły, specyfikowane pod konkretną grę komputerową.

Perspektywy rozwoju

W dzisiejszych czasach przytłaczająca większość gier komputerowych udostępnia możliwość rozrywki sieciowej. Odpowiedzią na to zapotrzebowanie powinna być taka budowa silnika, aby umożliwiał on pracę z innymi instancjami gry na różnych komputerach, w architekturze *klient-serwer*. Drugim powodem takiej budowy aplikacji jest zapewnianie ochrony przed piractwem komputerowym. Wiele współczesnych gier, w szczególności *MMORPG* jest projektowana w wyżej wymienionej architekturze. Do uruchomienia gry wymagane jest połączenie z Internetem oraz weryfikacja, czy uruchamiana gra nie przedstawia się kluczem gry uruchomionej na innym komputerze, przez innego użytkownika.

W procesie dalszego rozwoju gry warto jest wykonać implementację edytora poziomów, czyli zintegrowanego z silnikiem narzędzia pozwalającego na importowanie plików zasobów poprzez GUI, edycję sceny gry lub nawet edycję obiektów. Aplikacja spełniałaby rolę tzw. *Projektanta* (ang. Designer) znanego z środowiska .NET czy aplikacji *QTCreator*. Oprócz wspomnianych funkcji zadaniem aplikacji *Projektanta* byłoby generowanie kodu tworzącego obiekty na scenie, zgodnie z oknem graficznego projektowania sceny oraz kompilacja całości do pliku wykonywalnego.

Ostatnim, stałym punktem rozwoju jest podążanie za rozwojem kart graficznych. Coraz większe możliwości pozwalają na poprawę jakości wyświetlanej grafiki przez wprowadzanie różnych efektów czy bardziej realistycznych tekstur. Należy jednak pamiętać, że duża popularność starszych gier świadczy o fakcie, iż gra powinna być przede wszystkim *grywalna*, a więc być efektem dobrego pomysłu, niekoniecznie genialnej grafiki.

Krytyczne spojrzenie

Powstała biblioteka została w pełni zaprogramowana przez jedną osobę, a proces implementacji odbywał się niespełna rok. Tak krótki czas zmuszał mnie do okrojenia projektu z kilku funkcjonalności które w mojej ocenie warte były zaimplementowania. Brak poniższych elementów, nie znalazł się jednak w rozdziale w którym zostały opisane perspektywy rozwoju, z powodu większej wagi tych funkcjonalności. Przedstawione poniżej braki powinny być potraktowane priorytetowo w dalszych pracach nad systemem.

Implementacja biblioteki została wykonana pod platformę *Windows* i jest kompatybilna od wersji XP wzwyż. System operacyjny *Windows* został wybrany przez wzgląd na jego dominującą pozycję i popularność wśród użytkowników. Dalszymi perspektywami rozwoju powinno być zaimplementowanie warstwy pośredniej między silnikiem a systemem operacyjnym dla Mac OS X oraz Linux. Ciekawym obszarem dalszych prac może być również dostosowanie silnika do API systemów iOS oraz Android które wykorzystują OpenGL.

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Kolejną słabą stroną silnika jest ograniczony do dwóch kontrolerów system sterowania rozgrywką. W dzisiejszych czasach popularne stały się również pady, znane z konsol do gier, które wyparły używane dawniej joysticki. Dla podniesienia realistyki gier wyścigowych nie można zapomnieć o implementacji kontrolera - kierownicy.

Ostatnim wielkim nie obecnym jest interfejs skryptowy do silnika. W niektórych grach autorzy wyspecyfikowali API programowania interfejsów graficznych oraz programowania dodatków do gry. Przykładem udostępnienia interfejsu do gry na wielką skalę jest gra *World of Warcraft* firmy *Blizzard*. W sieci istnieje mnóstwo dodatków (ang. addons) umożliwiających zmianę wyglądu graficznego interfejsu użytkownika (ang. UI), ale też prowadzenia statystyk z przebiegu gry jak aktywność poszczególnych graczy, wartość obrażeń jakie zadają czy siłę leczenia. Udostępnienie graczom modyfikacji interfejsu gry, czyni ją ciekawszą i zapewnia większe zainteresowanie, co przekłada się na zysk firmy.

Ostatnim, elementem jaki chciałbym wymienić jest wsparcie dla szerszej gamy plików modeli trójwymiarowych, w tym opisujących animacje. Animowane ruchy modeli są niezbędne do tworzenia współczesnych gier. Brak modułu wczytywania animacji miał znaczący wpływ na wybór implementowanych gier. Trudno wyobrazić sobie popularną *strzelankę* (ang. shooter) czy grę strategiczną czasu rzeczywistego (ang. RTS), bez animowanych ruchów postaci czy elementów gry. Fakt ten bardzo ogranicza obecne możliwości silnika.

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

Bibliografia

[1] http://unity3d.com/
[2] Will Goldstone. Projektowanie gier w środowisku Unity 3.x. Helion, 2012. ISBN 978-83-246-3984-7.
[3] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides. Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku. Helion, 2010. ISBN 978-83-246-2662-5.
[4] http://qt-project.org/doc/qt-5.0/qtcore/qobject.html
[5] http://www.pegi.info/pl/index/id/366/
[6] James F. Kurose, Keith W. Ross. Sieci komputerowe. Ujęcie całościowe. Wydanie V. Helion, 2010. ISBN 978-83-246-2632-8. Aplikacje z obszaru P2P, s. 189-200.
[7] http://qt-project.org/doc/qt-5.0/qtcore/signalsandslots.html
[8] http://msdn.microsoft.com/en-us/library/5zwkzwf4(v=vs.110).aspx
[9] http://help.adobe.com/pl_PL/ActionScript/3.0_ProgrammingAS3/WS5b3ccc516d4f bf351e63e3d118a9b90204-7e54.html
[10] Mark DeLoura. Perełki programowania gier. Vademecum profesjonalisty. Helion, 2002. ISBN 83-7197-704-2. System profilujący działający w czasie rzeczywistym, s. 139-150.
[11] http://docs.unity3d.com/Documentation/Manual/Profiler.html
[12] Jerzy Grębosz. Symfonia C++ Standard. Programowanie w języku C++ orientowane obiektowo. Helion, 2013.
[13] Andrew S. Tanenbaum. Systemy operacyjne. Helion, 2010. s 182-188.
[14] http://unity3d.com/unity/workflow/scripting
[15] https://www.allegro.cc/
[16] http://www.libsdl.org/
[17] http://qt-project.org/doc/qt-4.8/objecttrees.html
[18] http://mst.mimuw.edu.pl/lecture.php?lecture=gk1&part=Ch8
[19] http://zach.in.tu-clausthal.de/teaching/cg_literatur/ lighthouse3d_view_frustum_culling/index.html
[20] http://rastergrid.com/blog/2010/10/gpu-based-dynamic-geometry-lod/
[21] http://lua.org.pl/5.1/manual.html
[22] http://qt-project.org/doc/qt-4.8/demos-interview-main-cpp.html
[23] http://www.qtcentre.org/wiki/index.php?title=Singleton_pattern
[24] http://www.cplusplus.com/reference/functional/
[25] http://sourceforge.net/p/predef/wiki/OperatingSystems/
[26] http://msdn.microsoft.com/enus/library/windows/ desktop/ms633573%28v=vs.85%29.aspx
[27] http://www.opengl.org/wiki/OpenGL_and_multithreading
[28] http://msdn.microsoft.com/enus/library/windows/ desktop/ms682052%28v=vs.85%29.aspx
[29] http://msdn.microsoft.com/en-us/library/windows/desktop/ms682052(v=vs.85).aspx
[30] http://visibleearth.nasa.gov/view.php?id=73934
[31] http://wyborcza.pl/1,75517,59660.html
[32] http://cpp0x.pl/kursy/Kurs-OpenGL-C++/Optymalizacja/244
[33] http://www.eg-models.de/formats/Format_Obj.html
[34] http://www.fileformat.info/format/bmp/egff.htm
[35] http://msdn.microsoft.com/en-us/library/windows/desktop/ee872027%28v=vs.85%29.aspx
[36] http://cpp0x.pl/kursy/Kurs-OpenGL-C++/Selekcja-obiektow/234

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29

[37] http://www.statista.com/statistics/276601/number-of-world-of-warcraft-subscribers-by-quarter/
[38] http://msdn.microsoft.com/en-us/library/vstudio/d5zhxt22.aspx
[39] http://msdn.microsoft.com/en-us/library/ms680657%28VS.85%29.aspx
[40] http://msdn.microsoft.com/en-us/library/windows/desktop/ms680360%28v=vs.85%29.aspx
[41] http://www.debuginfo.com/articles/effminidumps.html
[42] http://cpp0x.pl/kursy/Kurs-OpenGL-C++/Test-zaslaniania/241
[43] http://www.blender-models.com/model-downloads/vehicles/sci-fi/id/small-spaceship/

Zawartość CD

Na załączonej do pracy płycie CD zostały umieszone poniższe dane:

- W katalogu "Dokumentacja" został umieszczona dokumentacja kodu silnika Doxygen oraz Latex.
- W katalogu "Gry" zostały umieszczone pliki binarne trzech gier oraz ich zasoby.
- W katalogu "Praca" został umieszczony niniejszy dokument.
- W katalogu "Projekt" zostały umieszczone wszystkie pliki źródłowe oraz plik solucji Visual Studio 2012.

Spis rysunków

Rys. 1 Diagram klas modułu Rdzeń Silnika 18
Rys. 2 Diagram klas modułu Zarządca zdarzeń 24
Rys. 3 Diagram klas modułu Zasoby 27
Rys. 4 Diagram klas modułu Renderowanie 30
Rys. 5 Maska, tło oraz wynik mieszania 33
Rys. 6 Diagram klas modułu Scena 34
Rys. 7 Diagram klas modułu Narzędzia 42
Rys. 8 Mapa wysokości świata 44
Rys. 9 Mapowanie pikseli obrazu na wierzchołki terenu 46
Rys. 10 Sposób generowania siatki trójkątów 47
Rys. 11 Mapa wysokości z nałożoną teksturą 50
Rys. 12 Rozciąganie tekstury na stromych zboczach 50
Rys. 13 Gra warcaby. Instrukcja 51
Rys. 14 Gra warcaby. Skalowanie *damki* 53
Rys. 15 Gra wyścig. Kadr z gry 55
Rys. 16 Gra wyścig. Proces powstawania terenu 56
Rys. 17 Gra wyścig. Zastosowanie drzewa ósemkowego do ukrywania niewidocznych obiektów 58
Rys. 18 Gra snajper. Ekran wczytywania 59
Rys. 19 Gra snajper. Widok broni 61
Rys. 20 Gra snajper. Widok przez lunetę 62

Pobrano z https://repo.pw.edu.pl/ / Downloaded from Repository of Warsaw University of Technology 2026-07-23 12:29